StreamingBandit : Experimenting with Bandit Policies

Kruijswijk, Jules

2020-08

Kruijswijk , J , van Emden , R , Parvinen , P & Kaptein , M 2020 , ' StreamingBandit : Experimenting with Bandit Policies ' , Journal of Statistical Software , vol. 94 , no. 9 , pp. 1-47 . https://doi.org/10.18637/jss.v094.i09

http://hdl.handle.net/10138/324462
https://doi.org/10.18637/jss.v094.i09

Journal of Statistical Software

August 2020, Volume 94, Issue 9. doi: 10.18637/jss.v094.i09

StreamingBandit: Experimenting with Bandit Policies

Jules Kruijswijk
Tilburg University

Robin van Emden
Jheronimus Academy of Data Science

Petri Parvinen
University of Helsinki

Maurits Kaptein
Tilburg University

Abstract

A large number of statistical decision problems in the social sciences and beyond can be framed as a (contextual) multi-armed bandit problem. However, it is notoriously hard to develop and evaluate policies that tackle these types of problems, and to use such policies in applied studies. To address this issue, this paper introduces **StreamingBandit**, a Python web application for developing and testing bandit policies in field studies. **StreamingBandit** can sequentially select treatments using (online) policies in real time. Once **StreamingBandit** is implemented in an applied context, different policies can be tested, altered, nested, and compared. **StreamingBandit** makes it easy to apply a multitude of bandit policies for sequential allocation in field experiments, and allows for the quick development and re-use of novel policies. In this article, we detail the implementation logic of **StreamingBandit** and provide several examples of its use.

Keywords: sequential decision-making, multi-armed bandit, data streams, sequential experimentation, Python.

1. Introduction

In the canonical multi-armed bandit (MAB) problem a gambler faces a number of slot machines, each with a potentially different payoff. It is the gambler's goal to make as much profit (or, in the case of gambling, as little loss) as possible by sequentially choosing which machine to play, learning from the observations as she goes along (Whittle 1980; Berry and Fristedt 1985). The gambler faces a trade-off between exploration and exploitation (Macready and Wolpert 1998): on the one hand she wishes to play the machine that was successful in earlier attempts as often as possible (exploitation), but on the other hand she wishes to find the machine with the highest payoff through experimentation (exploration). The MAB problem, and its generalization, the contextual MAB (or CMAB) problem – in which before selecting

a machine the gambler observes the state of the world that could be related to the optimal choice of machine at that point in time – provides a flexible formalization for studying sequential treatment-allocation procedures in the social sciences and beyond (Dudík *et al.* 2011a; Li, Chu, Langford, and Wang 2011; Agrawal and Goyal 2013).

A multitude of policies addressing (contextual) decision problems have been conceived and evaluated (see, e.g., Berry and Fristedt 1985; Chapelle and Li 2011; Dudík *et al.* 2011a). Indeed, the randomized controlled trial (RCT, or ϵ-first in the literature on sequential decision-making; Chapelle and Li 2011) is in itself a specific policy devised to address the exploration-exploitation trade-off in which an exploration phase, the trial itself, is followed by exploitation. Other policies range from simple heuristics such as "play the winner" (Lachin, Matts, and Wei 1988; Villar, Bowden, and Wason 2015) to asymptotically optimal policies such as upper confidence bound (UCB) methods (Auer and Ortner 2010; Garivier and Cappé 2011; Audibert, Munos, and Szepesvári 2009), and Bayesian methods such as Thompson sampling (Thompson 1933; Chapelle and Li 2011; Agrawal and Goyal 2012). It is difficult to assess which of these policies performs best in distinct applied problems, however, due to the omission of the counterfactuals in the (field) evaluations of a policy (Li *et al.* 2011): one does not know what the outcome would have been had another choice been made anywhere along the sequence of decisions. Hence the data resulting from an evaluation can often not be used to evaluate alternative policies. To evaluate a range of possible policies one has to resort to either simulation methods – which often lack external validity due to the large number of assumptions encoded in the simulation – or to recent offline evaluation methods (Li *et al.* 2011; Agarwal *et al.* 2016). Offline methods provide the opportunity to obtain unbiased estimates of the performance of different policies on historical data, but these approaches are only practically feasible when the number of choice alternatives is relatively low and/or the number of sequential choices is large. Furthermore, the assumptions that justify these methods – such as stationarity and a non-zero probability for each possible treatment at each interaction (Li *et al.* 2011) – are rarely fully justified in practice.

Despite these difficulties, effective (contextual) decision policies are potentially of great use in many areas. To unleash this potential researchers need to be able quickly to implement and evaluate distinct bandit policies in the field. This can be achieved by allowing substantive researchers easily to test different sequential allocation schemes. If easy-to-use software were available for evaluating and disseminating novel policies, such policies – which are actively being developed (e.g., Eckles and Kaptein 2014; Osban and Roy 2015; Bastani and Bayati 2020) – would be within reach of a broader research community. It is to this end that we developed **StreamingBandit**: an open-source RESTful Web application that allows researchers to formalize their sequential-allocation procedure as a CMAB problem and, by virtue of this formalization, easily to experiment with different policies.

In the remainder of this section we first engage in a high-level discussion of the basic usage of **StreamingBandit**, discuss related approaches, and provide an overview of the application and its installation. In Section 2, we describe the application in more detail, and demonstrate the setup and evaluation of a single policy. Here we also discuss the use of **StreamingBandit** for offline policy evaluation and we offer a number of performance measures. In Section 3, we introduce a number of currently implemented "default" policies and discuss methods of combining multiple policies. We detail two practical applications of **StreamingBandit** in Section 4, and finally in Section 5 we briefly discuss future work directions.

1.1. Basic usage

The basic setting we consider is the following. Consider an experimenter who interacts with the environment. At each interaction t:

1. the experimenter observes a context x_t,
2. subsequently, the experimenter chooses an action a_t,
3. and finally a reward r_t is observed.

The main aim of the experimenter is to maximize the cumulative reward $\sum_{t=1}^{N} r_t$ where N denotes the total number of interactions. To do so, the experimenter applies a policy π which is some function that takes the context x_t and the historical interactions, and returns an action. For convenience we denote all historical interactions using $\mathcal{D}_{(t-1)}$ and thus we have $\pi(x_t, \mathcal{D}_{t-1}) \rightarrow a_t$.

This sequential decision-making scheme is encountered in many real-life situations:

- *Personalized healthcare:* A physician meets with patients sequentially. For each patient, she observes a number of background characteristics (gender, age, current condition) constituting the context. Subsequently, her action is to choose a treatment such that the reward – measured in terms of the general health of the patient – is maximized.
- *Online advertising:* In online advertising a firm selecting an ad observes the context consisting of a description of the current user visiting a specific webpage. The action is to choose an advertisement from of a set of possible advertisements (possibly dependent on the context), and the rewards constitute the clicks on the ad.
- *Product-recommendation systems:* The context denotes all that is known about the user at a certain point in time. The action is choosing one of a set of products, and the reward consists of the revenue generated at each interaction.
- *Social-science experiments:* Many social-science experiments constitute a special case of contextual decision-making: participants are recruited sequentially during the experiment. The context consists of all that is known about the participant, and sequentially the action is to assign a participant to a specific experimental condition (possibly dependent on the context in cases of stratified sampling, for example). Finally, the reward(s) consist of the outcome measures of the experiments.

The above list illustrates the generality of our approach: **StreamingBandit** can be used to allocate actions in all of the above applications.

To ensure the computational scalability of **StreamingBandit** we assume that, at the latest interaction $t = t'$, all the information necessary to choose an action can be summarized using a limited set of parameters denoted $\theta_{t'}$, the dimensionality of θ_t often being (much) smaller than that of $\mathcal{D}_{t-1}$. Given this assumption, we identify the following two steps of a policy:

1. *The decision step:* In the decision step, using $x_{t'}$ and $\theta_{t'}$, and often using some (statistical) model relating the actions, the context, and the reward, which is parametrized by $\theta_{t'}$, the next action $a_{t'}$ is selected. Making a request to **StreamingBandit**'s `getaction`

REST endpoint returns a JSON object containing the selected action. Optionally, the probability $p_{t'}$ of selecting this action (the `propensity`) and/or an identifier for this specific request (the `advice_id`), both of which are explained in more detail below, is also returned.

2. *The summary step:* In each summary step $\theta_{t'}$ is updated using the new information $\{x_{t'}, a_{t'}, r_{t'}, p_{t'}\}$. Thus, $\theta_{t'+1} = g(\theta_{t'}, x_{t'}, a_{t'}, r_{t'}, p_{t'})$ where $g()$ is some update function. Effectively, all the prior data, $\mathcal{D}_{t-1}$ are summarized in $\theta_{t'}$. This choice means that the computations are bounded by the dimension of θ and the time required to update θ instead of growing as a function of t. Note that this effectively forces users to implement an online policy (Michalak, DuBois, DuBois, Wiel, and Hogden 2012) as the complete dataset $\mathcal{D}_{t-1}$ is not revisited at subsequent interactions. Making a request to **StreamingBandit**'s `setreward` endpoint containing a JSON object including either the `advice_id` or a complete description of $\{x_{t'}, a_{t'}, p_{t'}\}$, and the reward $r_{t'}$, allows one to update $\theta_{t'+1}$ and subsequently to influence the actions selected at $t' + 1$.

For the basic usage of **StreamingBandit** the experimenter – or rather an external server or mobile application – sequentially executes requests to the `getaction` and `setreward` endpoints, and allocates actions accordingly. Using this setup, **StreamingBandit** can be used to sequentially select advertisements on webpages, for example, allocate research subjects to different experimental conditions in an online experiment, or sequentially optimize the feedback provided to users off a mobile eHealth application. We provide a number of practical examples in Section 4.

1.2. Related approaches

Theoretically, contextual decision-making relates to a broad literature ranging from active learning (e.g., Beygelzimer, Hsu, Langford, and Zhang 2010; Hanneke 2014) to the general setting of reinforcement learning (Sutton and Barto 1998; Szepesvári 2010). The contextual MAB problem (Dudík *et al.* 2011a; Li, Chu, Langford, and Schapire 2010; Agarwal, Hsu, Kale, Langford, Li, and Schapire 2014) we consider here is a specific instance of reinforcement learning: it is a problem that is well-studied both without contextual information (Berry and Fristedt 1985) and in numerous generalizations, such as the continuous bandit (Mandelbaum 1987) and bandits with dependencies (Pandey, Chakrabarti, and Agarwal 2007). The current work also relates to recent discussions on offline policy evaluation (Dudík, Erhan, Langford, and Li 2012; Dudík, Langford, and Li 2011b), although it is distinct from the multi-world testing service presented by Agarwal *et al.* (2016) in its focus on running (adaptive) policies online versus the online collection of data combined with the offline evaluation of policies. The field is too large to be properly reviewed in this paper, and we refer the reader to Schwartz, Bradlow, and Fader (2017) and the references therein for an accessible introduction and contemporary applications.

Here we narrow our discussion of related approaches to related software projects, which we split into the following four categories: (i) software for A/B testing, (ii) software for general (supervised) learning, (iii) software for offline policy evaluation, and (iv) software for (sequential) optimization. The first category relates to our current project in that A/B tests – or randomized experiments – are used in many fields to address (C)MAB problems: one devotes a (pre-set) number of interactions to random exploration, after which the best performing action is selected and further exploited. This approach has become standard in many

web companies (Jiang, Shi, Shang, Geng, and Glass 2016). A more advanced version, often referred to as "multi-variate testing" runs many A/B tests in parallel, possibly exploiting a factorial structure between the actions. Several commercial systems, such as Google Analytics, provide A/B testing abilities (Google 2018), Optimizely (Optimizely 2017), and Mixpanel (Mixpanel 2017).

An effective policy depends heavily on the ability to predict the next reward given a context. Once available, a (large) dataset of contexts, actions, and rewards constitutes a supervised learning problem. Many general supervised learning solutions have been developed recently, such as **CNTK** (Seide and Agarwal 2016), **GraphLab** (Collet, Sassolas, Lhuillier, Sirdey, and Carlier 2016), **GeePS** (Cui, Zhang, Ganger, Gibbons, and Xing 2016), **MLlib** (Meng *et al.* 2016), **TensorFlow** (Abadi *et al.* 2016), and **Minerva** (Reagen *et al.* 2016). Some of these, such as **Vowpal Rabbit** (Langford, Li, and Strehl 2011) and **Jubatus** (Hido, Tokui, and Oda 2013), explicitly include libraries implementing specific bandit policies, or evaluation methods for bandit policies on existing, offline, data sets. Specific software projects for offline policy evaluation, and hence the ability to evaluate policies on existing datasets, are also available (see, e.g., Komiyama, Honda, and Nakagawa 2015; Nugent 2015; **Striatum** Contributors 2016). Others have provided language-specific code libraries implementing different policies, although most of these efforts seem to be a) geared towards computer scientists and experienced developers and b) not focused on field deployment (see Kaufmann, Cappé, and Garivier 2012a; Galbraith 2016; Sola 2015, and the references therein).

There are a number of platforms that allow for sequential optimization: Google Analytics (Google 2018), for example, supports Thompson sampling (Agrawal and Goyal 2012; Kaptein 2014; Thompson 1933), which is a method for sequentially allocating visitors to different actions dynamically based on the observed outcomes. However, contextual knowledge is not included. Yelp **MOE** (Yelp 2014) is an open-source software package that implements optimization over a large parameter space via sequential A/B tests in which Bayesian optimization is used to compute parameters for the next best A/B test. Finally, the Decision Service (Agarwal *et al.* 2016) implements a number of functionalities implemented by **StreamingBandit** using a similar formalization (the summary and decision steps). This software package focuses on continuously collecting data to update and deploy policies that are evaluated offline, whereas **StreamingBandit** focuses on evaluating (adaptive) policies online.

1.3. An overview of streaming bandit API calls

StreamingBandit is a Python 3 (Van Rossum *et al.* 2011) application that runs a **Tornado** web server (The **Tornado** Authors 2016) and discloses a REST API that facilitates the implementation of the summary and decision steps described above. A user of **StreamingBandit** first creates an experiment and subsequently implements – or adopts based on the library of available policies – a policy using Python 3. A policy specification consists of a) some code implementing the decision step given $\theta_{t'}$ and $x_{t'}$, and b) some code implementing the summary step given the observed outcomes to update $\theta_{t'}$. Figure 1 presents an overview of the architecture of **StreamingBandit**. The application discloses a number of REST endpoints to facilitate the creation and editing of experiments and the extraction of data from running experiments. All endpoints apart from the `getaction` and `setreward` require the user to authenticate using a secure cookie. Logging in can be done by passing a JSON object to the `login` endpoint containing the parameters `username` and `password`; if the username and password are valid a

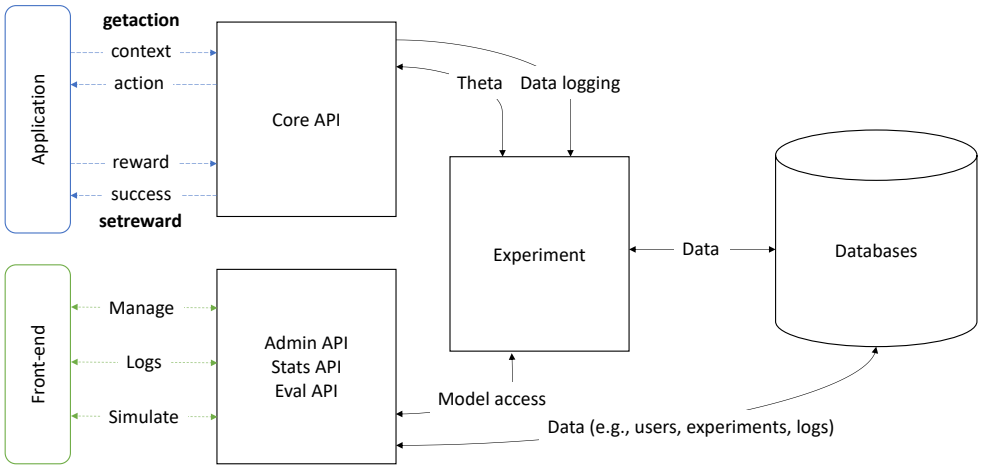

Figure 1: High-level architecture of **StreamingBandit**.

secure-cookie is returned. New users can be created using the `user` call and posting the relevant information. For convenience, we provide a separate UI (a separate software project that can be found at `https://github.com/Nth-iteration-labs/streamingbandit-ui`) that allows easy point-and-click administration and management of experiments. Here we detail the primary endpoints and describe their functionality. We have already introduced the `getaction` and `setreward` calls, of which the full specification is:

GET `getaction`: The query-string parameters consist of the experiment identification number, `exp_id` (string), a `key` (string), and the `context` (JSON). The call executes the decision step of a policy associated with the `exp_id` and returns an `action` (JSON), which optionally contains the elements `advice_id` (string), and `propensity` (float). The `key` is used to authenticate the request.

GET `setreward`: The query-string parameters consist of the `exp_id`, the `key`, the `reward` (JSON) and either the `advice_id`, in which case the `context` and `action` are retrieved from the associated `getaction` call, or the the `context` and `action` themselves. Subsequently, the summary step of the policy associated with the associated `exp_id` is executed and a JSON object containing the status is returned.

The primary endpoints at which to manage the experiments are:

GET `exp`: Returns a JSON object listing the `exp_id` and `name` of each experiment.

POST `exp`: Posting a JSON object containing the parameters `name`, `getcontext`, `getaction`, `getreward` and `setreward` creates a new experiment. The last four fields should contain executable restricted Python 3 code. To ensure some safety in the executed code we limit the functionality of these customer scripts to a subset of Python 3 code, using self-defined built-ins. This will disallow, for instance, the import of any other packages apart from the one we already make available. It also means that the user does not need to import any packages into the code because they are made available in the

built-ins before any code is executed. The code in the `getaction` and `setreward` fields implements the decision and summary steps, respectively. The `exp` endpoint accepts a number of optional parameters, which we detail in Section 2.1. A valid POST request to the `exp` endpoint returns a JSON object containing the `exp_id` and the `key` of the newly created experiment.

The code in the `getcontext` and `getreward` fields is not strictly necessary; these two snippets of code provide for the opportunity to simulate sequential decisions. This is extremely useful for debugging and can be used in simulation studies of a policy. Passing the query-string parameter `n` (int, default=1) to rest endpoint `eval/<exp-id>/simulate` sequentially executes the `getcontext`, `getaction`, `getreward` and `setreward` code of the associated experiment n times.

PUT `exp/<exp_id>`: If the `exp_id` string in the url is a valid experiment for the current user, this call edits the existing experiment. The parameters are the same as those used for creating experiments.

GET `exp/<exp_id>`: Returns the name and `getaction` and `setreward` code for a specific experiment.

DELETE `exp/<exp_id>`: Deletes an experiment. When an experiment is deleted all the user-generated settings are removed, as well as the current θ. However, logged data associated with the experiment is maintained.

GET `exp/<exp_id>/resetexperiment`: Resets the experiment: the current state of θ is deleted, but all the other information is retained and the policy can still be executed.

Next to these administrative calls, the application provides a number of calls to monitor running experiments and retrieve logged data.

GET `stats/<exp_id>/currenttheta`: Returns the current θ for the experiment as a JSON object.

GET `stats/<exp_id>/summary`: Returns an overview of the number of requests to the `getaction` and `setreward` endpoints.

GET `stats/<exp_id>/rewardlog`: Returns the logged `setreward` events (including the `context`, `action`, and `reward` objects) for the current experiment. It can be used for offline policy evaluation (see, e.g., Li *et al.* 2011; Agarwal *et al.* 2016). The `limit` (int) query-string parameter limits the dump to the last k events.

GET `stats/<exp_id>/actionlog`: Returns all the `getaction` events for the current experiment. Again, the `limit` parameters limit the dump to the last k events.

GET `stats/<exp_id>/log`: Returns a JSON file of all data that was explicitly logged by the user using `self.log()` in the policy specification of an experiment.

Requests made to non-existing REST endpoints result in a 404 status error, whereas erroneous calls to existing end-points return a JSON object containing a key `error` with an informative error message.

Implemented policies: "defaults"

StreamingBandit comes with a number of implemented policies to tackle standard (contextual) decision problems. A JSON object containing a list of defaults can be retrieved using the endpoint `default`, and calling `default/<default_id>` gives the code for a specific default. We have implemented the following policies, amongst others:

- ϵ-first: Implements the standard randomized clinical trial approach to the (C)MAB problem: the first $t < n$ interactions, where n is set by the user, are allocated to actions randomly, after which the action with the highest average reward is selected for the remaining interactions.
- ϵ-greedy: Implements a greedy policy in which a proportion p of interactions is randomly allocated to the available actions, whereas a proportion of $(1-p)$ interactions is allocated to the action with the highest average reward at that point in time.
- Thompson sampling for the k-armed Bernoulli bandit: Thompson sampling provides a Bayesian solution to the MAB problem (Thompson 1933; Agrawal and Goyal 2012). We implement Thompson sampling for the Bernoulli bandit (e.g., $r \in \{0, 1\}$). Thompson sampling allocates actions proportional to one's current belief – as quantified using a posterior distribution – that an arm is optimal (Kaptein 2014).
- Lock-in feedback: Lock-in feedback is an allocation scheme for dealing with continuous actions ($a \in \mathbb{R}$) in which small systematic oscillations in the action choice over time are used to derive the gradient of the reward function and take a step toward the (local) maximum of that function (see Kaptein, Emden, and Iannuzzi 2016a; Kaptein, Van Emden, and Iannuzzi 2016b, for details).
- Bootstrap Thompson sampling: Bootstrap Thompson sampling provides a computationally appealing alternative to Thompson sampling in cases in which it is hard to sample directly from the posterior distribution of a model online (see Eckles and Kaptein 2014). In essence, the posterior distribution is approximated using an online bootstrap distribution (Owen and Eckles 2012).

We provide examples of the use of these policies in Section 3. **StreamingBandit** is easily extended and new defaults can be added by adding code to the `/resources/defaults` folder of the application in a folder with an informative name that contains the following four files:

1. `get_context.py`: A Python script that generates a JSON object encoding a context.
2. `get_action.py`: A script that takes a JSON object encoding the context, and returns a JSON object containing the action.
3. `get_reward.py`: A script that generates a reward using a context and action JSON.
4. `set_reward.py`: A script that takes a context, action, and reward JSON and handles the logic of updating θ.

Restarting the web application after adding these files will automatically include the novel policy in the list of defaults. We welcome submissions of new default policies and other implementations. See Section 1.5 for more details.

StreamingBandit libraries

StreamingBandit was created to quickly create and test alternative policies in the field. This can be done by altering the `getaction` and `setreward` codes associated with an experiment. However, given that a number of operations are often encountered in the online processing of incoming data, **StreamingBandit** also provides a number of Python modules:

- `base`: This module provides functionalities for online (row-by-row) updates of, e.g., counts, means, variances, proportions, and covariances.
- `lm`: Implements an online version of a linear regression model.
- `bts`: Takes a model (e.g., `lm`) and a row of data and produces (or updates) an online bootstrap distribution of the parameters.
- `lif`: Implements the lock-in feedback policy, as described in Kaptein and Ianuzzi (2016).
- `thompson`: Implements Thompson sampling for the k-armed Bernoulli bandit, amongst others.
- `thompson_bayes_linear`: Implements model-based Thompson sampling using a Bayesian linear regression model.

New modules can be added to the application by adding a script to `/libs`. For detailed documentation of the individual modules we refer the reader to `http://nth-iteration-labs.github.io/streamingbandit/libs.html`.

1.4. Installation, deployment, and documentation

The **StreamingBandit** source code is available from `https://github.com/Nth-iteration-labs/streamingbandit/` and the documentation can be accessed on `http://nth-iteration-labs.github.io/streamingbandit/`. There are several ways in which **StreamingBandit** can be used:

1. At `http://sb.nth-iteration.com` we provide a running instance of **StreamingBandit**. You apply for a user account by sending an email to the corresponding author of this paper, and use our hosted webserver for (small-to-medium-sized) projects.
2. The easiest way to get going independently is probably to use our **Docker** container (Merkel 2014). The following commands assume that you have **docker** and **docker-compose** installed, and that you are inside a folder in which you wish to put the source-code of **StreamingBandit**[1]. If so, starting **StreamingBandit** requires, first, pulling the repository to your local system and going inside the folder:

   ```
   $ git clone http://github.com/Nth-iteration-labs/streamingbandit/
   $ cd streamingbandit
   ```

   Next, once you are inside the folder with all the source code, we can launch **Streaming-Bandit** by running:

[1]For more information on how to get started with **Docker**, see `https://docs.docker.com/get-started/`.

```
$ docker-compose up -d
$ docker exec -t streamingbandit_web_1 python3 ../insert_admin.py -p \
> test
```

The first command makes sure that all necessary containers, including the databases, are running. The second command creates a user account admin with the password "test". To gracefully stop and start the container after running the first command, run the following command:

```
$ docker-compose stop
$ docker-compose start
```

Starting the service will make **StreamingBandit** available at http://localhost:8080 or the **Docker**-set IP address.

Note that the above commands only start the back-end REST service. The following commands are also needed to launch our front-end:

```
$ docker-compose -f docker-compose.yml \
> -f docker-compose.front-end.yml up -d
$ docker exec -t streamingbandit_web_1 python3 \
> ../insert_admin.py -p test
```

The start and stop commands now change slightly as well:

```
$ docker-compose -f docker-compose.yml \
> -f docker-compose.front-end.yml stop
$ docker-compose -f docker-compose.yml \
> -f docker-compose.front-end.yml start
```

which starts and stops both the front-end and the back-end at the same time. The front-end can be reached at http://localhost or the **Docker**-set IP address.

The front-end source-code can be found in a separate repository at https://github.com/Nth-iteration-labs/streamingbandit-ui, but for this use-case it is not necessary to download the repository to your local system because we have uploaded a **Docker** image to the internet and **Docker** will download that image automatically via the `docker-compose` command.

3. For larger-scale projects we recommend installing from source and perhaps using a load-balancer. For details, please consult the documentation at http://nth-iteration-labs.github.io/streamingbandit/.

1.5. Further development

The above sections give the essential details of **StreamingBandit**. We gladly accept any contributions towards making **StreamingBandit** better and more useful. The guidelines for contributing to the development of **StreamingBandit** can be found in the documentation.

2. Getting started

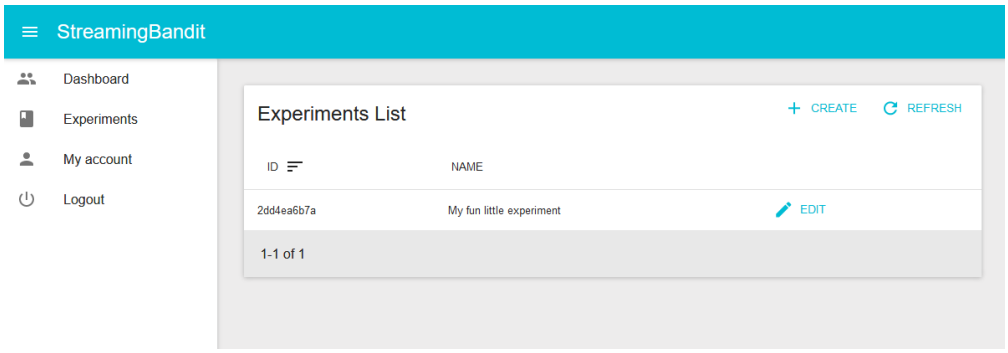

Figure 2: Screen shot of the default front-end for **StreamingBandit**.

In the remainder of this article we assume that the reader is running the default **Docker** container installation of **StreamingBandit**, and is using the management front-end for the administration of experiments. In introducing the details of setting up a policy we describe the setup and usage of a simple – but very frequently used – policy: ϵ-first. The code for this section and the following Sections 2.1, 3.1, and 3.2 is supplied with the package in the default experiments. It is called ϵ-first. When this policy is executed a sample of size n interactions is uniformly randomly allocated to a control ($a = \text{control}$) or treatment ($a = \text{treatment}$) action (or condition), after which the treatment is adopted if it is more effective than the control condition. With slight abuse of the notation this can be denoted:

$$\pi_{\texttt{epsilon-first}}(\mathcal{D}, n) = \begin{cases} a_t \sim \texttt{random}(\text{control}, \text{treatment}) & \text{if } t \leq, n \\ a_t = \texttt{max}(\bar{r}_{\text{control}}, \bar{r}_{\text{treatment}}) & \text{otherwise,} \end{cases} \tag{1}$$

where $\bar{r}_{\text{control}}$ denotes the sample average of outcomes observed in the control condition when $t \leq n$, and the last line denotes selection of the action with the highest empirical average reward when $t > n$. The management front-end – of which Figure 2 shows a screenshot – makes it easy to create a new experiment or to use one of the defaults as a starting point for creating one's own policies. We present the front-end in more detail in Appendix A. Once the experiment has been created it receives an `exp_id` and a key `key`. This enables the REST endpoints

```
http://HOST/getaction/<exp_id>?key=<key>&context={}
```

and

```
http://HOST/setreward/<exp_id>?key=<key>&context={}&reward={}&action={}.
```

The actual functionality is provided by the `getaction` and `setreward` code specified when the experiment is created, whereas the `getcontext` and `getreward` codes are useful for simulations and testing. Below we detail each of these in turn for the version of ϵ-first implemented in the defaults. Before that we should note that we will denote a few variables and functions using `self` inside the code. These variables and functions are denoted with `self` because they are part of the experiment class in which the custom code runs. For the most part, we will only use a reference to `self` with the following variables and functions:

- `self.context`
- `self.action`
- `self.reward`
- `self.get_theta()`
- `self.set_theta()`

The code for a simple ϵ-first implementation is as follows:

- `getcontext`: The canonical ϵ-first strategy does not consider a context. Hence, we leave this blank.
- `getaction`: The implementation of the decision step of ϵ-first is:

```
n = 100
mean_list = base.List(self.get_theta(key = "treatment"),
    base.Mean, ["control", "treatment"])
if mean_list.count() >= n:
    self.action["treatment"] = mean_list.max()
    self.action["propensity"] = 1
else:
    self.action["treatment"] = mean_list.random()
    self.action["propensity"] = 0.5
```

This code uses a number of libraries implemented in **StreamingBandit**: below we detail each line in turn. First, the sample size of the experiment, n in Equation 1, is set. The next line of code generates a list of `base.Mean` objects. This object provides the functionality to compute streaming updates of sample averages, and the list contains one such average for each of the possible treatments specified by name, using `["control", "treatment"]`. The `self.get_theta()` call is used to retrieve $\theta_{t'}$, which in this case thus contains two `base.Mean` objects named "control" and "treatment". A count, n, and mean reward, $\bar{r}$, are contained within each `base.Mean` object.

The resulting `mean_list` object thus, in this case, contains two `base.Mean` objects, each of which contains a mean value and a count that can be updated and manipulated. In the next lines the total count of the number of observations over all mean elements in the list is retrieved. If this is larger than n, the treatment with the highest average value is returned, otherwise a random element of the list is returned. The returned JSON object when making a call to `http://HOST/<exp_id>/getaction?key=<key>` and filling in the correct `exp_id` and `key` appears as follows:

```
{"action":
     {"treatment": "control",
     "propensity": 0.5},
"context": {}}
```

where the value of `treatment` changes randomly as long as $n \leq t$.

- `getreward`: Rewards can be simulated by using a few lines of Python 3 code

```
if self.action["treatment"] == "control":
    self.reward["value"] = np.random.normal(4, 1)
else:
    self.reward["value"] = np.random.normal(6, 2)
```

  in which the rewards for the selected action in the decision step are drawn from a normal distribution ($r_{\text{control}} \sim \mathcal{N}(4, 1)$, $r_{\text{treatment}} \sim \mathcal{N}(6, 2)$).

- `setreward`: When a reward has been generated, the summary step for the ϵ-first policy is implemented as:

```
n = 100
mean_list = base.List(self.get_theta(key = "treatment"),
    base.Mean, ["control", "treatment"])
if mean_list.count() < n:
    mean = base.Mean(self.get_theta(key = "treatment",
        value = self.action["treatment"]))
    mean.update(self.reward["value"])
    self.set_theta(mean, key = "treatment",
        value = self.action["treatment"])
```

  which again uses the **libs.base** library. After this the action is retrieved and the associated mean object is updated using `mean.update` as long as the exploration phase is ongoing. The last line stores $\theta_{t'+1}$ such that it can be retrieved again for future decision-making. In this implementation, after the experiment when $n > t$, θ is no longer updated. Note that a slightly more elaborate version of this example that facilitates propensity scores (see Section 2.1) can be found in the defaults (see Section 1.3).

As stated above, the `getcontext` and `getreward` codes are not strictly necessary to use the implemented policy in field studies; these two snippets of code merely provide the opportunity to simulate an experiment, a feature that is extremely useful for debugging. In actual evaluations of a policy the data resulting from these calls would be sent by the outside world (e.g., via a website or mobile application). However, to demonstrate the utility of the `getcontext` and `getreward` codes, note that a request to the endpoint `/eval/<exp_id>/simulate` with parameters `N=150`, `seed=1271246`, and `verbose=False` yields the following JSON response:


```
{
    "theta": {
        "treatment:control": {
            "n": "52",
            "m": "4.0259030511640885"
        },
        "treatment:treatment": {
            "n": "48",
            "m": "5.829777419810004"
        }
    },
```

```
    "simulate": "success",
    "experiment": "121e3e0aeb"
}
```

which shows the number of times the `treatment` and `control` conditions were selected (`n`) and their respective mean reward (`m`). Although we simulated 150 interactions, the total number of interactions stored in θ is $48 + 52 = 100$ because in the implementation above we stop updating θ when $t > n$.

2.1. Additional features

We described the setup and simulation of a simple bandit experiment in the previous section. The description skipped over a number of useful features of **StreamingBandit**, which we address below.

Offline analysis of bandit policies

When we first introduced the `getaction` endpoint we mentioned the optional return field `propensity`. In a number of default policies, the return object contains this propensity p_t, which is the probability of selecting the action at interaction t. By way of an illustration, for ϵ-first, as detailed above, the computation of p_t is as follows:

$$p_t = \begin{cases} 0.5 & \text{if } t \leq n \\ 1 & \text{otherwise} \end{cases}$$

Whenever it is possible to compute these propensities – which is sometimes difficult, such as when $a \in \mathbb{R}$ – the default policies include p_t. This serves two purposes:

1. When addressing contextual sequential decision problems, and when the probability of selecting an action depends on the context, the propensity p_t can be used for inverse propensity matching or weighting (Austin 2011) to improve the estimate of the causal effect of the action by accounting for the contextual covariates (see, e.g., Imbens and Rubin 2015; Pearl 2009).

2. When p_t is included, the logged data of an experiment can be used for the offline evaluation of alternative decision policies. This can be attained by using inverse propensity scoring (ips). Suppose we are evaluating a policy π using a logged dataset containing N events. The ips estimate of average reward of the policy can be obtained by computing

$$\texttt{ips}(\pi) \quad = \quad \frac{1}{N} \sum_{t=1}^{N} \mathbb{1}\{\pi(x_t) = a_t\} r_t / p_t$$

   where the indicator is 1 when the action of π matches the action in the logs. Agarwal *et al.* (2016) provide a more extensive discussion of the benefits of using offline methods to evaluate alternative policies.

Advice ID, delayed rewards, and logging

When we described the [POST] `exp` endpoint we omitted a number of optional parameters that can be supplied in the JSON object. First of all, we skipped discussion of the `advice_id`

parameter. This Boolean indicates whether or not the `getaction` call should return an `advice_id`. When set to `True` the `advice_id` parameter enforces a direct link between the `getaction` and `setreward` endpoints. In the example discussed above we were implicitly assuming that the application consuming the REST API would handle the logic that ensures that by the time the `setreward` endpoint is called, the `context`, `action` (including the `propensity`), and `reward` are properly supplied. However, this could be challenging for some consuming applications. In such cases, setting `advice_id = True` would require the consuming application to merely specify the `advice_id` when making a request to the `setreward` endpoint; **StreamingBandit** will merge the actions and context that were provided earlier in the associated `getaction` call with the rewards supplied in the `setreward` call.

When setting `advice_id = True`, one can also specify a) how long the `advice_id` will be retained (in hours). This is useful in some specific applications. In an online advertising experiment, for example, when a click on an advertisement is not registered within 12 hours it is extremely unlikely that this will happen in the future; it is more likely that the appropriate call to the `setreward` with $r_t = 0$ failed to register. Setting `delta_hours=12` and `default_reward = {"reward":"0"}` ensures that after twelve hours the `setreward` call associated with the `advice_id` is automatically executed with a reward of zero. It should also be noted that although all the examples provided in this paper sequentially execute the `getaction` and `setreward` calls, this is not at all a necessity. However, any bias in a (learning) model that might originate from, e.g., a delay in the arriving data in the `setreward` calls should be explicitly handled by the user.

Finally, we have not yet discussed the `hourly_theta` Boolean: if this is set to `True` when creating the experiment, the state of θ will be logged every hour. Calling `stats/<exp_id>/hourlytheta` with parameter `limit` returns the last k of these snapshots of θ, which could be useful for monitoring the progress of an experiment over time.

The nesting of policies

In addition to the libraries described earlier, and the methods for storing and retrieving data `self.get_theta()` and `self.set_theta()`, there are a number of methods available to the user from the code supplied in the `getaction` and `setreward` fields. The most interesting of these is the ability to instantiate other experiments within a running experiment. By way of illustration, the code

```
experiment = Experiment(exp_id =  <exp_id>)
self.action = experiment.run_action_code(context =  self.context)
```

can be used to run the `getaction` code of the experiment with `exp_id=<exp_id>` from another experiment. Similarly, `experiment.run_reward_code()` would execute the `setreward` code for another experiment. This allows the user to nest different experiments, and hence to essentially use a sequential decision policy π^* to decide from among a range of policies that are being executed $\pi_{1,\ldots,k}$. We provide a working example of this policy nesting in the Section 3.6.

2.2. Performance

To examine the performance of our RESTful API we set up an Ubuntu 16.04 x64 quad-core virtual server with 16GB of RAM running the **StreamingBandit** server, and additionally

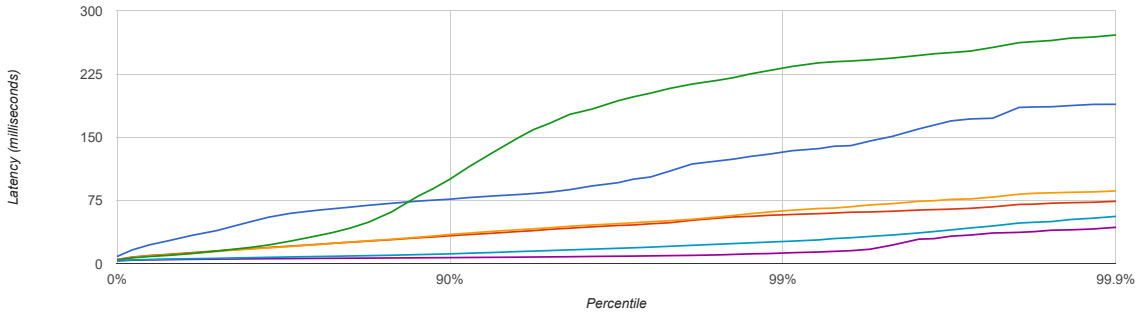

Figure 3: Latencies of basic **Tornado** calls when taxed by **wrk2** at a maximum throughput of 100 (Tornado_R100) versus 500 (Tornado_R500) calls per second (cps), as compared to **StreamingBandit** "AB test" (AB_GetSet_R100, throughput limited at 100 cps) and empty `getaction/setreward` calls (with Empty_GetSet_100, Empty_GetSet_200, Empty_GetSet_300 at respectively 100, 200 and 300 cps).

installed the **wrk2** load generator on a smaller (single core, 1GB RAM) Ubuntu 16.04 x64 machine connected to the same subnet within the same datacenter. We chose **wrk2** (Tene 2015) as our load generator, as it is a HTTP benchmarking tool that is capable of generating significant load when run on a single CPU, and can easily be extended to test different RESTful HTTP methods through the use of `Lua` (Ierusalimschy 2016) scripts.

To ensure that our load tests would not be hampered by OS related limitations we optimized `sysctl.conf` on both machines, turning off disk swapping, upping the number of connections per port, and optimizing port reuse. We also tested our client-server throughput with **iPerf3** (The **iPerf** Authors 2016). These tests indicated a throughput of 736 Mbits/s – more than enough bandwidth to safeguard against system-level I/O bottlenecks interfering with our API-level tests.

On completion of our test-bed we proceeded to run several **wrk2** load tests, focusing on industry-standard API performance measures (De 2017). The results for a single **wrk2** thread running 100 concurrent AB test `getaction` calls at a time with a throughput limit of 1000 requests per second were the following:

- Average, max and standard deviation of latency: 21.09ms, 90.56ms, 13.22ms
- Throughput, in requests per second: 100 (equal to max set **wrk2** throughput)
- Top total CPU utilization: 69% (Of which: Python 3 65% of one of four available CPU's)
- Top Heap memory utilization: 3%

When we compared these numbers against some representative Python web framework benchmarks (Klenov 2015) we found that **StreamingBandit** could hold its own. Still, to obtain a more objective measure of how "empty" versus "AB test" **StreamingBandit** `getaction` calls measure up to basic, vanilla **Tornado** requests, we compared these as well. The results, as illustrated in Figure 3, demonstrate that **StreamingBandit** adds little overhead to basic **Tornado** processing, and scales well up to 250 to 300 requests per second when running on a single virtual CPU core. The relatively minor increment in throughput and latency between

the "empty" and the "AB test" experiments further indicates that **StreamingBandit** offers sufficient capacity to implement more complex experiments.

3. Examples of the implemented policies

In the following we work out a number of different (C)MAB policies. First, we present a simple implementation of ϵ-greedy (Sutton and Barto 1998), then we introduce Thompson sampling for the canonical k armed Bernoulli bandit (Thompson 1933), and for optimal design in between-subject experiments (Kaptein 2014). We proceed by demonstrating two possible policies to deal with the continuum-bandit problem (problems in which $a \in \mathbb{R}$): Bootstrap Thompson sampling for a CMAB problem using a simple linear model (Eckles and Kaptein 2014) and lock-in feedback (LIF, Kaptein and Ianuzzi 2016). We further demonstrate how **StreamingBandit** can be used to nest multiple policies, and show how **StreamingBandit** can be used to evaluate multiple policies in parallel using the offline evaluation method proposed by Li *et al.* (2011). This latter approach is, to the best of our knowledge, novel. All of the implementations discussed in this section can be found in the defaults (see Section 1.3).

3.1. E-greedy

One frequently used policy is called ϵ-greedy (Sutton and Barto 1998). It is implemented in a simple problem consisting of a control and a treatment arm, as we considered when we introduced ϵ-first, by playing the arms uniformly randomly with some probability ϵ, and selecting the hitherto best-performing arm with probability $1 - \epsilon$. The same `getcontext` and `setreward` codes as in our ϵ-first example above are used to implement ϵ-greedy, as follows:

- `getaction`:

```
e = .1
mean_list = base.List(self.get_theta(key = "treatment"),
    base.Mean, ["control", "treatment"])
if np.random.binomial(1,e) == 1:
    self.action["treatment"] = mean_list.random()
    self.action["propensity"] = 0.1*0.5
else:
    self.action["treatment"] = mean_list.max()
    self.action["propensity"] = (1-e)
```

  Where, contrary to our ϵ-first example, we explicitly include the computation of the propensity p_t.

- `setreward`: The summary step for the ϵ-first can be implemented as:

```
mean = base.Mean(self.get_theta(key = "treatment",
    value = self.action["treatment"]))
mean.update(self.reward["value"])
self.set_theta(mean, key = "treatment",
    value = self.action["treatment"])
```

which is the same as for ϵ-greedy except for the fact that the respective means are updated at each interaction t instead of $n < t$.

Running a simulation with $n = 1000$ and `seed` $= 1271246$ gives:


```
{
    "theta": {
        "treatment:control": {
            "n": "70",
            "m": "4.011771491758239"
        },
        "treatment:treatment": {
            "n": "930",
            "m": "5.9609857188253015"
        }
    },
    "simulate": "success",
    "experiment": "3ea45886b5"
}
```


in which it is clear that the `treatment` arm is preferred.

3.2. Thompson sampling for the K-armed Bernoulli bandit

As our second example we provide the code to implement Thompson sampling for the classical Bernoulli bandit problem where the rewards are either 0 or 1, and for each arm $k = 1, \ldots, K$ the probability of success (reward $= 1$) is μ_k (Kaufmann, Korda, and Munos 2012b). Thompson sampling is a Bayesian policy in which one selects an action with a probability that is proportional to one's posterior belief that the action is optimal (see Kaufmann *et al.* 2012b, for details). In the Bernoulli reward case the Beta(α, β) distribution provides a convenient a priori choice in that after observing a Bernoulli trial the posterior distribution is simply Beta$(\alpha + 1, \beta)$ in the case of success, and Beta$(\alpha, \beta + 1)$ in the case of failure. Using S_k and F_k to denote the number of failures and successes for arm k, both of which are 0 at the start, Thompson sampling proceeds as follows; at each interaction t,

1. for each arm $k = 1, \ldots, K$, sample $d_k(t)$ from Beta$(S_k + 1, F_k + 1)$,
2. select arm $k(t) = \arg\max_k d_k(t)$,
3. and if $r_t = 1$ then $S_k = S_k + 1$ or when $r_t = 0$ then $S_k = S_k + 1$.

Thompson sampling for the 4-arm Bernoulli bandit problem can be implemented as follows:

- `getcontext`: The Bernoulli bandit does not consider a context; we leave this field blank.
- `getaction`: The decision step, using the **libs.thompson** library, can be implemented using:

```
propl = thmp.BBThompsonList(self.get_theta(key = "treatment"),
    ["1","2","3","4"])
self.action["treatment"] = propl.thompson()
self.action["propensity"] = propl.propensity(self.action["treatment"])
```

where the four arms are indexed using the numbers $1 - 4$.

- `getreward`: Bernoulli rewards can be simulated using:

```
self.reward["value"] = np.random.binomial(1,
    (0.2*int(self.action["treatment"])))
```

which produces Bernoulli rewards with a probability of $0.2, 0.4, 0.6, 0.8$ for the four arms respectively.

- `setreward`: Finally, the updates of the posterior distributions are implemented using

```
prop = base.Proportion(self.get_theta(key = "treatment",
    value = self.action["treatment"]))
prop.update(self.reward["value"])
self.set_theta(prop, key = "treatment",
    value = self.action["treatment"])
```

Running a simulation with $n = 1000$ and `seed` $= 1271246$ gives:


```
{
    "theta": {
        "treatment:1": {
            "p": "0.14285714285714288",
            "n": "7"
        },
        "treatment:4": {
            "p": "0.8025404157043878",
            "n": "866"
        },
        "treatment:2": {
            "p": "0.20000000000000004",
            "n": "10"
        },
        "treatment:3": {
            "p": "0.5982905982905986",
            "n": "117"
        }
    },
    "experiment": "1de9753f51",
    "simulate": "success"
}
```


Which demonstrates that arm 4 is clearly, and correctly, preferred.

3.3. Thompson sampling for optimal design

Another example that could have practical relevance in social-science experiments is presented in Kaptein (2014): when running an experiment comparing two groups that receive different treatments, assuming unequal variances in the observed continuous outcomes, it is beneficial to allocate a larger number of subjects to the treatment with the highest variance to increase the precision in the obtained effect-size estimate. The Thompson sampling policy to implement this sequential allocation is to compute – using a normal-inverse χ^2 model – the posterior variances σ_1^2 and σ_2^2 of the two treatments in the summary step. Next, in the decision step, a draw d from each of the two posterior distributions σ_1^2 and σ_2^2 is obtained and the treatment is selected for which $\frac{d}{n}$, where n denotes the number of subjects allocated to the respective treatment, is highest. This choice leads to the largest reduction in the estimated standard error of the mean difference between the two groups. We refer the interested reader to Kaptein (2014) for details. This sequential allocation scheme can be implemented In **StreamingBandit** using:

- `getcontext`: Left blank as no context is considered
- `getaction`: In the summary step, we retrieve a list of two variance objects one for each treatment. Variance objects, and the ability to update these online, are included in **base** library. Next, we implement Thompson sampling on the level of the posterior variances of the outcomes; this is included in the **libs.thompson** library:

```
varList = thmp.ThompsonVarList(self.get_theta(key = "treatment"),
    ["control","treatment"])
self.action["treatment"] = varList.experimentThompson()
```

- `getreward`: To simulate outcomes with unequal variances we can use:

```
if self.action["treatment"] == "control":
    self.reward["value"] = np.random.normal(0, 1)
else:
    self.reward["value"] = np.random.normal(1, 5)
```

- `setreward`: And finally, we update the respective posterior variance when new observations arrive:

```
var = base.Variance(self.get_theta(key = self.action["treatment"]))
var.update(self.reward["value"])
self.set_theta(var, key = "treatment", value = self.action["treatment"])
```

Running a simulation with $n = 100$ and `seed` $= 43123$ gives:


```
{
    "theta": {
        "treatment:treatment": {
            "s": "1453.3754330265062",
            "n": "77",
            "x_bar": "0.777831868342291",
```

```
            "v": "19.123360960875083"
        },
        "treatment:control": {
            "s": "32.31094303640007",
            "n": "23",
            "x_bar": "0.032257238191552844",
            "v": "1.4686792289272759"
        }
    },
    "experiment": "84b4d7eda",
    "simulate": "success"
}
```


This result highlights two things: First, it is clear that the treatment condition with the highest variance is indeed selected more often. This is the expected behavior to ensure that the precision of the estimate is increased. Second, the result demonstrates the internals of the `base.Variance` object: to compute a variance in a data stream we maintain a count (`n`), a mean (`m`), and the numbers `s` and `v`; of these `v` is the current sample variance, whereas `s` in an auxiliary variable used to implement Welford's method for computing a variance online (Welford 1962).

3.4. Bootstrap Thompson sampling

Bootstrapped Thompson sampling (BTS) is a recent approach devised to address CMAB problems (see, e.g., Eckles and Kaptein 2014; Osban and Roy 2015). The basic idea behind BTS is that instead of using a draw from the posterior distribution of the parameters of interest to decide on the next allocation, as is the case in previous Thompson sampling examples, one can maintain, online, a number of bootstrapped estimates of the parameters. These bootstrapped estimates can then be used to balance exploration and exploitation by randomly selecting one of the bootstrap replicates (see Eckles and Kaptein 2014, for details).

StreamingBandit implements this sequential allocation scheme quite generally using the double-or-nothing bootstrap (Owen and Eckles 2012). The appeal of BTS compared to traditional Thompson sampling is that a) it can be fully carried out online as long as the point estimates of interest can be obtained online, and b) it can be used in many situations in which obtaining draws from the true posterior density of interest is computationally difficult. Here we provide a simple example of the implementation of BTS using a linear model to relate the actions, the contexts, and the rewards.

For ease of exposition, let us consider a practical example. Suppose we are concerned with choosing a `price` (the action) of a product sold online such that the `revenue` is maximized (the reward). Let us further assume that we believe the relation between these two quantities is quadratic, and that we think the optimal sales price differs between `new` customers and `returning` customers. The following code implements this scenario such that it can be simulated:

- `getcontext`: The get context code simulates the visit of either a new or a returning visitor.

  ```
  self.context["customer"] = random.choice(["new", "returning"])
  ```

- `getaction`: Next, the get action code, which is slightly more involved, uses the **lm** library to instantiate $m = 100$ linear models of the form

$$\begin{aligned}\texttt{revenue} \sim \quad & \beta_0 + \beta_1 \texttt{price} + \beta_2 \texttt{price}^2 + \beta_3 \texttt{new} + \\ & \beta_4 \texttt{price} * \texttt{new} + \beta_5 \texttt{price}^2 * \texttt{new}.\end{aligned}$$

  Here, the starting values of the model β's are initially set to zero. The `BTS` object maintains $m = 100$ of these models, whereas the remaining code samples one of these $m = 100$ models and computes the `price` that maximizes the expected revenue given the current customer and the current state of the parameters. We add comments to the code to improve readability:

```
# Instantiate BTS with m=100 samples:
BTS = bts.BTS(self.get_theta(), lm.LM, m = 100, default_params = \
    {'b': np.zeros(6).tolist(), 'A' : np.identity(6).tolist(), 'n' : 0})

# Return one of the m samples:
model = lm.LM(default = BTS.sample())

# Retrieve its coefficients:
betas = model.get_coefs()

# Create dummy for customer
if(self.context["customer"] == "returning"):
    customer = 1
else:
    customer = 0

# Maximize the function
if betas[2] != 0 or betas[5] != 0:
    x = ( (-(betas[1] + betas[4] * customer)) /
        (2*(betas[2] + betas[5] * customer)) )
    x = np.asscalar(x)
    if x < 5 or x > 20:
        x = np.random.uniform(5,20)
else:
    x = np.random.uniform(5,20)

# Return the price
self.action["price"] = x
```

  Note that we restrict the prices to be between 5 and 20, such that if `BTS` needs some more exploration, it will not go towards extreme values, which may happen if a linear model is selected that has no parabola – in a field experiment you might want not have your prices restricted to certain ranges as well.

- `getreward`: In the get reward code we use a logistic function to simulate the probabilities of accepting or rejecting the product at the offered price for different customer types.

```
# Get parameters
# Create dummy for customer
if(self.context["customer"] == "returning"):
    customer = 1
else:
    customer = 0
price = self.action["price"]

# Create logistic function
logistic = lambda x: 1 / (1 + numpy.exp(-x))

# Compute purchase yes / no
buy = numpy.random.binomial(1,
    logistic(-0.1 * (price - (10+4*customer))**2))

# Compute the reward
self.reward["revenue"] = buy * price
```

Here it is clear that new customers are more inclined than returning customers to buy for higher prices, the revenue-maximizing price being ≈ 10.9 for new customers, and ≈ 14.7 for returning customers.

- `setreward`: Finally, after generating the reward, the summary step for this policy can be implemented as follows:

```
# Extract values:
# Create dummy for customer
if(self.context["customer"] == "returning"):
    customer = 1
else:
    customer = 0
price = self.action["price"]

# Create feature vector and response:
X = [1, price, price**2, customer, customer*price, customer*price**2]
y = self.reward["revenue"]

# Instantiate the m = 100 lm models
BTS = bts.BTS(self.get_theta(), lm.LM, m = 100, default_params = \
    {'b': np.zeros(6).tolist(), 'A' : np.identity(6).tolist(), 'n' : 0})

# Update the model parameters using the new observation
BTS.update(y, X)

# Store the updated values
self.set_theta(BTS)
```

To illustrate the outcomes of this sequential allocation scheme we run a simulation with

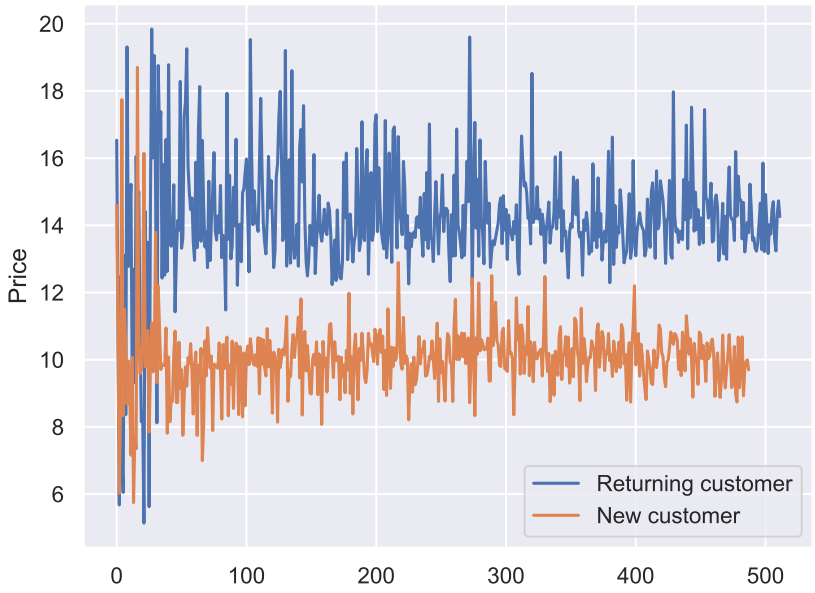

Figure 4: Overview of the selected prices of BTS with $m = 10$ and $N = 1000$ for both returning and new customers (separate lines).

$N = 1000$ and `seed` $= 43123$ setting the "log results" to `True`. Next, using the logged data, we plot the selected prices for each of the customer types separately. Figure 4 shows the progression of the recommended prices for each customer type; it is clear that these display a lot of exploration behavior early in the data stream, but after about 100 observations the BTS policy seems to exploit more and settles on a price that is close to the maximum in a large number of the interactions.

3.5. Lock-in feedback

Picking a price was considered the intended action in the previous example. Hence, in this case $a_t \in \mathbb{R}$. This so-called continuum bandit problem (Bubeck, Munos, and Stoltz 2011) has many practical applications. Here we provide an example of an alternative strategy for selecting the actions in such a setting. The term "lock-in feedback" has been coined for this policy, which is described in detail in Kaptein and Ianuzzi (2016). The basic idea of the policy is to oscillate the values of the actions at a known frequency and to amplify this frequency in the observed rewards. Next, the noise can be integrated out, which produces a result that – given mild assumptions regarding the function relating the reward and the action, which we denote $r = f(a)$ – is directly proportional to the first derivative of $f()$. Subsequently, this first derivative can be applied, using a gradient-ascent-type algorithm, to move a step towards the maximum of $f()$.[2]

Lock-in feedback is appealing because the experimenter does not need to specify $f()$ explicitly – as we did in the previous example – and the allocation policy has proved to be robust in cases of concept drift (e.g., a situation in which $f()$ changes over time). Lock-in feedback can

[2]Note, however, that lock-in feedback does not attain asymptotically optimal performance due to its constant exploration.

be implemented as follows:

- `getcontext`: For the sake of simplicity we consider a case without contextual information.

- `getaction`: The implementation of the decision step lock-in feedback is relatively simple using the **lif** library:

```
theta = self.get_theta(all_float = False)
Lif = lif.LiF(theta, x0 = 3.0, a = 0.5, t = 20, gamma = 0.02,
    omega = 1.0, lifversion = 1)
suggestion = Lif.suggest()
self.action["x"] = suggestion["x"]
self.action["t"] = suggestion["t"]
self.action["x0"] = suggestion["x0"]
```

  where we refer to the `lif` documentation at http://nth-iteration-labs.github.io/streamingbandit/ for details regarding the parameters of the `lif` method.

- `getreward`: Rewards can be simulated as follows:

```
x = self.action["a"]
self.reward["r"] = -1 * pow((x - 5), 2)
```

  where clearly the highest reward is obtained when $a = 5$.

- `setreward`: Finally, the summary step can be implemented using

```
theta = self.get_theta(all_float = False)
Lif = lif.LiF(theta, x0 = 3.0, a = 0.5, t = 20, gamma = 0.02,
    omega = 1.0, lifversion = 1)
Lif.update(self.action["t"], self.action["x"], self.reward["r"],
    self.action["x0"])
self.set_theta(Lif)
```

Running a simulation with $n = 1000$ and `seed` $= 43123$ gives:


```
{
    "experiment": "2c070b0c17",
    "simulate": "success",
    "theta": {
        "x0": "4.9885573624026183",
        "t": "1000",
        "Yw":
        "[[981.0, 5.32735516395185, -0.0364325832561265],
        [982.0, 4.8625703653723935, 0.0023573471591685955],
        [983.0, 4.512596828979011, 0.11280696009422152],
        [984.0, 4.599313553861246, 0.06234374986452693],
        [985.0, 5.0430128577187805, -0.00010219719086955388],
        [986.0, 5.435840666435192, -0.0851018295535888],
```

```
        [987.0, 5.416689112014516, -0.07446606113542824],
        [988.0, 5.00321866842203, -1.599797646219837e-07],
        [989.0, 4.575643921281427, 0.07422661156312545],
        [990.0, 4.527110675559689, 0.10305907582127445],
        [991.0, 4.902338977243727, 0.0008184672724298606],
        [992.0, 5.356344978818639, -0.046745403226967665],
        [993.0, 5.471845324305438, -0.10767081744630806],
        [994.0, 5.142633922830094, -0.00314257165629084],
        [995.0, 4.671518435948291, 0.034171390252964014],
        [996.0, 4.491614635071014, 0.128372350733584],
        [997.0, 4.7684753555595965, 0.011794462261213126],
        [998.0, 5.247511963923335, -0.01586222111864476],
        [999.0, 5.488454341053454, -0.11925205127092639],
        [1000.0, 5.26974690054797, -0.020460304127878852]]"
    }
}
```

Where `Yw` and `t` are internals used to execute the policy, and `x0` represents the current location of the search algorithm; initialized at 3.0 it is, with a value of 4.988 after $t = 1000$ observations, indeed close to the actual maximum of 5.

The **libs.lif** library has already been applied successfully in various settings, as described, for example, in a recent paper investigating the use of the LIF algorithm to optimize scenarios in behavioral economics (Kaptein *et al.* 2016a), and in another paper in which LIF is applied to the optimization of the physical features of an avatar in multiple dimensions in response to a continuous stream of ratings, provided by the participants of the experiment (Kaptein *et al.* 2016b). In both settings, LIF proved admirably capable of finding, and locking into, optima – despite the considerable noise often inherent in such human-choice-related studies. Hence, **StreamingBandit** was used successfully in these settings to allocate, in real-time, experimental treatments to subjects in a social-science study.

3.6. Nesting of policies

A further interesting use of **StreamingBandit** relates to the ability to nest multiple policies; this allows the user to, e.g., use an ϵ-greedy strategy to decide between the use of lock-in feedback and BTS, as presented above. Here we provide an example of this nesting of policies in which we assume that the user has instantiated two experiments, one implementing ϵ-first as described in Section 2, and one implementing ϵ-first as described in Section 3.1. We can now setup a third experiment that allocates interactions to either of these two experiments by referring to their `exp_id`'s[3]. This can be achieved as follows:

- `getcontext`: We do not consider a context in this example
- `getaction`: Let us assume that we wish to uniformly randomly allocate half of our interactions to the ϵ-first experiment, and half of our interactions to the ϵ-greedy experiment. This can be done using:

[3]Note that including a non-existent `exp_id` leads to errors in running the code. **StreamingBandit** does not explicitly check for such errors inside the code of the user. We have implemented an implicit call that can be used to check if the experiment is valid by using `exp_nested.is_valid()`.

```
id1 = "275fc0a66" # The exp_id of E-First
id2 = "18aec502c2" # The exp_id of E-Greedy

choice = np.random.binomial(1, 0.5)

# Run the e-first experiment
if choice == 0:
    exp_nested = Experiment(exp_id = id1)
    self.action = exp_nested.run_action_code(context = {})
    # We return the experiment number for later use
    self.action["experiment"] = id1
    # We re-compute the propensity based on the probability of picking
    # the nested experiment
    self.action["propensity"] = self.action["propensity"] * 0.5
# or, run the e-greedy experiment
else:
    exp_nested = Experiment(exp_id = id2)
    self.action = exp_nested.run_action_code(context = {})
    self.action["experiment"] = id2
    self.action["propensity"] = self.action["propensity"] * 0.5
```

- `getreward`: Rewards can be simulated using the code we also introduced in Section 3.1.
- `setreward`: The summary step for these nested experiments can be implemented using:

  ```
  # Based on the exp_id we know which experiment to update
  exp_id = self.action["experiment"]

  exp_nested = Experiment(exp_id = exp_id)
  exp_nested.run_reward_code(context = self.context,
      action = self.action, reward = self.reward)
  ```

  Which simply, based on the supplied `exp_id`, updates the correct experiment.

Running a simulation with $n = 2$ and `seed` $= 13214$, and the output set to `verbose`, gives:


```
{
    "data": {
        "0": {
            "theta": {},
            "context": {},
            "reward": {
                "value": 4.439687566610595
            },
            "action": {
                "propensity": 0.45,
                "experiment": "18aec502c2",
                "treatment": "treatment"
```

```
            }
        },
        "1": {
            "theta": {},
            "context": {},
            "reward": {
                "value": 7.559583564021055
            },
            "action": {
                "propensity": 0.25,
                "experiment": "275fc0a66",
                "treatment": "treatment"
            }
        }
    },
    "experiment": "1c57b6d641",
    "simulate": "success"
}
```


Which shows that in the first interaction ϵ-greedy was selected, which subsequently selected the treatment arm, and in the second interaction ϵ-first was selected. Obviously, this functionality can be greatly extended to use any sequential decision policy to decide between any other policy. This nesting makes **StreamingBandit** a versatile tool; we illustrate a practical application of the nesting in Section 3.7.

3.7. Parallel evaluation of multiple policies

Whereas the nesting discussed in the previous section allows one to allocate different interactions to different policies, the example we provide here allows one to evaluate, using a measure of average reward for example, multiple bandit policies in parallel. The idea behind the parallel evaluation derives from recent work on the offline evaluation of bandit policies. Li *et al.* (2011) show that one can evaluate multiple bandit policies offline by simply running through an existing data set of actions and rewards obtained using uniform random selections of the actions. For each interaction t in the offline data set one uses a bandit policy to generate a proposal action a'_t, and if the randomly selected action at that point in time matches the proposal (thus $a'_t = a_t$), then the reward is used to update the estimated performance of the policy. If not, then the time point is discarded. This leads to an evaluation of the policy with an expected number of observations of $\frac{1}{k}T$, where k is the number of possible actions and T the total number of observations in the offline data set. Multiple offline evaluation runs can subsequently be used to estimate and compare the expected performance of different policies.

Here we extend this idea to the parallel evaluation of multiple bandit policies. The implementation in **StreamingBandit** to compare, in parallel, the performance of the ϵ-first and ϵ-greedy experiments as introduced above is surprisingly straightforward:

- `getcontext`: For simplicity we again consider an empty context.
- `getaction`: In the decision step an action is chosen at random:

```
self.action["treatment"] = random.choice(["control","treatment"])
```

- `getreward`: Rewards can again be simulated using the code we also introduced in Section 3.1.

- `setreward`: Finally, after generating a reward, the summary step for the parallel evaluation of the policies is given below, where we again insert comments in the code to improve readability:

```
# Create a list of experiments / policies to evaluate
policies = ["18aec502c2", # E-Greedy
    "275fc0a66"]  # E-First

# For each experiment
for exp_id in policies:

    # Initialize the experiment:
    exp_nested = Experiment(exp_id)

    # Compute the suggested action:
    suggestion = exp_nested.run_action_code(context = {})

    # See if the suggested action matches the actual action:
    if suggestion["treatment"] == self.action["treatment"]:

        # And if so store the performance of the policy:
        mean = base.Mean(self.get_theta(key = "policy_means",
            value = exp_id))
        mean.update(self.reward["value"])
        self.set_theta(mean, key = "policy_means", value = exp_id)

        # And finally update the policy:
        exp_nested.run_reward_code(context = {},
            action = self.action, reward = self.reward)
```

This code implements Algorithm 2 of Li *et al.* (2011).

Running a simulation with $n = 250$ and `seed` $= 43123$ using the above specification gives:

```
{
    "theta": {
        "policy_means:275fc0a66": {
            "m": "5.243151928222057",
            "n": "114"
        },
        "policy_means:18aec502c2": {
            "m": "6.050783848360902",
            "n": "114"
        }
```

```
    },
    "experiment": "270ed59474",
    "simulate": "success"
}
```

This output shows that, in this test run, the average reward of the ϵ-greedy policy is slightly higher than that of the ϵ-first policy. This is due to the fact that ϵ-first has a random exploration phase of $n = 100$. Since both policies now only have had 114 accepted actions, ϵ-first will have explored much more than ϵ-greedy and will choose the suboptimal action more, resulting in a lower average reward.

4. Applied usage

In this section, we describe some of the practical applications of **StreamingBandit**. First, we explore its use in assessing the effects of discounts in online selling; this small, initial trial highlights the simple use of **StreamingBandit** to collect data in-the-field. Second, we introduce its use in a social-science experiment.

4.1. Online marketing

StreamingBandit was used by an online cash-refund company to examine the effects of their pricing scheme. The company offers customers the opportunity to sign up for a refund program. After signing up they are provided with discounts, in the form of a cash refund, as long as their online purchases are carried out through the online platform. The refund company has negotiated different agreements with a large number of different e-commerce stores, and the discount percentages they have obtained vary from store to store. By default, the refund company offers half of its negotiated discount to the customer, and takes the other half as a fee for its services. However, it has no clear idea as to whether this 50/50 (or $\frac{1}{2}$) split is optimal in the sense that it maximizes its profit, which is influenced by the total number of purchases, the size of the purchases, and the way in which the negotiated discount is split between the company and the customer.

The company set up **StreamingBandit** to explore the effects of the different splits – in their definition running from 0 to 1 where 1 means that the total negotiated discount is fully passed on to the customer and 0 means that all of it is retained by the company – on their resulting profits. Here we present a simple implementation of the random exploration of different splits that the company carried out for a very small number of $n = 103$ unique customers in one specific store. The implementation was as follows:

- `getcontext`: Because this is a field exploration, the context was provided by the participating company. It consisted of a JSON object containing the `maxpercentage`, which contained the negotiated discount for the specific store that was viewed by a customer. It looked like this:

  ```
  {"context" : {"maxpercentage" : 8.5}}
  ```

  where the `maxpercentage` for the specific store from which our presented data originated was always 8.5%. However, our implementation described below is able to address

changing maximum percentage(s) between different stores. Note that this can be simulated in **StreamingBandit** using the following `getcontext` code:

```
self.context["maxpercentage"] = numpy.random.uniform(1,10)
```

- `getaction`: The implementation of the decision step was straightforward since the company initially set out merely to examine the effects of random fluctuations of the discounts offered. The implementation was as follows:

  ```
  maxpercentage = self.context['maxpercentage']
  split = np.random.uniform()
  discount = split * maxpercentage
  self.action['split'] = split
  self.action['discount'] = discount
  ```

  Here, first the `maxpercentage` is retrieved. Next, a split is computed with `split` $\sim$ `unif`$(0, 1)$, after which the percentage discount to be offered to the customer is computed and then both the `split` and the actual `discount` are returned in the `action` object.

- `getreward`: The online platform would display the computed `discount` to the visiting customer, and subsequently a reward would be generated by virtue of the customer's purchasing one or multiple products resulting in a `revenue`. The online platform returns both the `revenue` as well as the `split` and `discount`. This can be simulated using:

  ```
  self.action['split'] = self.action['split']
  self.action['discount'] = self.action['discount']
  self.reward['revenue'] = numpy.random.uniform(0,100)
  ```

- `setreward`: Finally, given that the aim of the company was merely to collect data on the effect of the changing splits, it did not need any `setreward` code because **StreamingBandit** automatically logs all the data that is received with a `setreward` call.

This simple implementation allowed the refund company to vary the split randomly (instead of using the current de-facto $\frac{1}{2}$ split) and to log the resulting revenue.

Figure 5 provides an overview of the relation between the suggested `split` and the resulting profit in euros of the refund company. The `profit` for the rebate-company is defined as the maximum discount percentage (8.5%) times one minus the split (between 0 and 1), times the revenue. Each dot represents one completed purchase by one customer (possibly containing multiple products). Note that we while limit the presented results here to a single e-commerce store, the store sells multiple products and hence the revenue per customer can vary greatly. It seems from the limited data of these $n = 103$ unique customers for a single store that a high customer-refund offer – but as a result a low margin for the company – leads to low profits, whereas an offer that is significantly below the current $\frac{1}{2}$ split increases the company's profits.

The company intends to use **StreamingBandit**, now that the software is integrated into its current online service, to experiment with different sequential allocation schemes that offer different splits between competing stores or between different customers. Using the random data and an adaption of the offline evaluation method developed by Li *et al.* (2011) (also described in Section 3.7), the company hopes to find the policy that has the best model fit on their data. Note that here every step towards solving this statistical decision problem

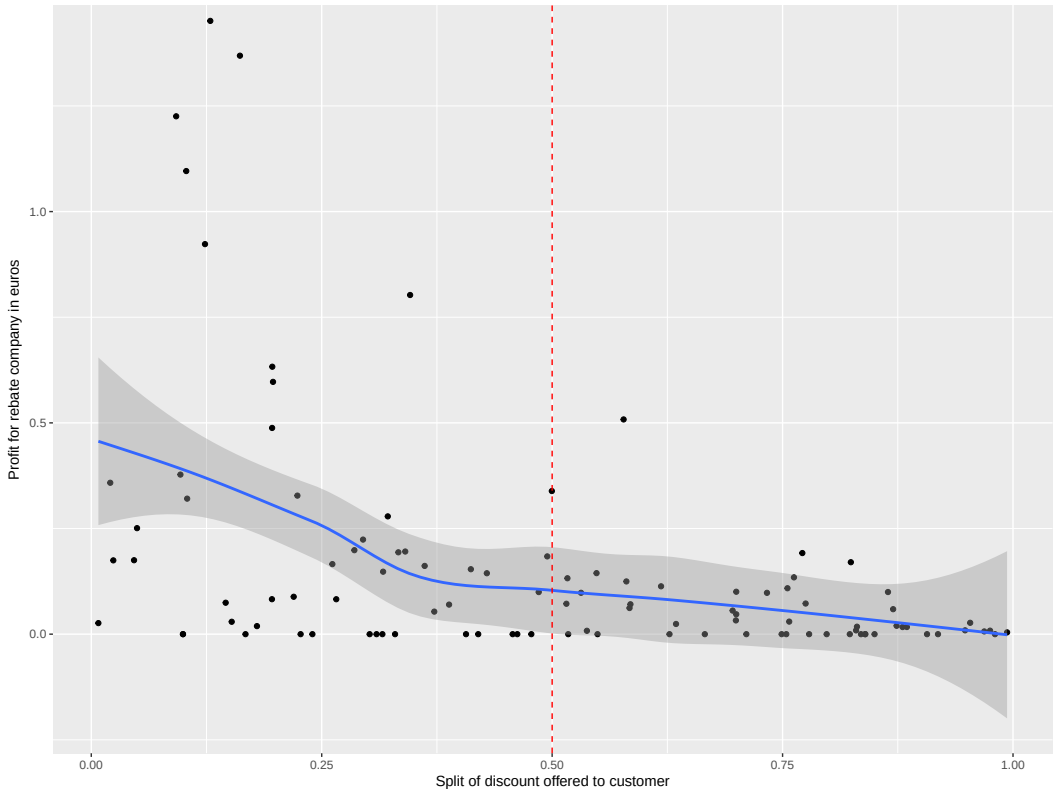

Figure 5: Overview of the effect of the offered split of the discount on the profit of the refund company in euros. Data collected using random selection of the refund percentage using **StreamingBandit**. The figure presents data on $n = 103$ unique customers. The dashed red line represents the company's current $\frac{1}{2}$ split.

involves using **StreamingBandit** – from gathering data, to policy evaluation, to the final, live setting. This provides a simple example of the utility of **StreamingBandit** for field trials of bandit policies.

4.2. Social science experiment

The second applied use of **StreamingBandit** we present concerns a social-science experiment examining the decoy effect (see Kaptein *et al.* 2016a, for a full description of the experiment). In short, the decoy effect states that people may be persuaded to switch from one offer to another by the presence of a third option (the decoy) that, rationally, should have no influence on the decision-making process. For example, when asked to choose between a laptop with a good battery but a poor memory and a laptop with a poor battery but a good memory, people seem to shift their preference between the two if the offer is accompanied by a third laptop, the decoy, that has a battery as good as the latter but an even worse memory, and hence should in any case be an irrelevant option. The placement of the decoy in the product-attribute space is heavily studied in the literature: researchers manipulate the exact battery life in hours and the RAM in GB of the decoy laptop, and study the resulting choices that people make.

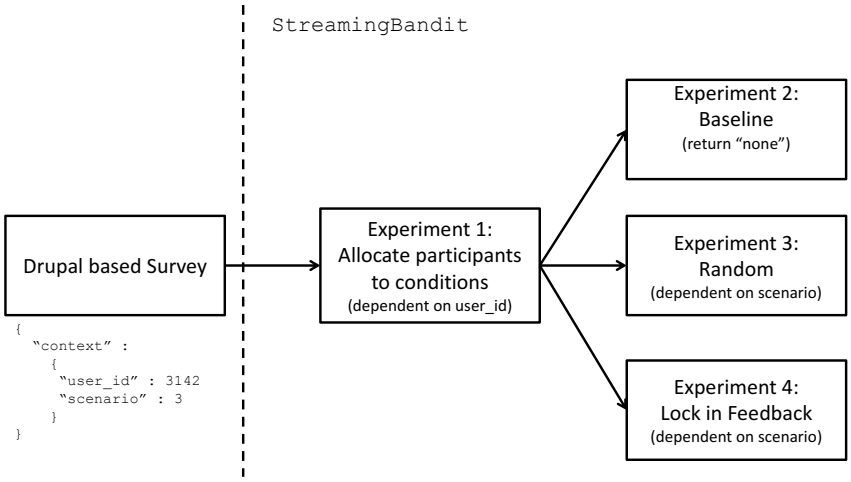

Figure 6: Schematic setup of the 4 **StreamingBandit** experiments used to realize the data-collection in (Kaptein *et al.* 2016a).

Kaptein *et al.* (2016a) used **StreamingBandit** to study whether lock-in feedback, the sequential optimization scheme introduced in Section 3.5, can be used to find the optimal placement of the decoy – only considering changes on one dimension. The authors considered not only the laptop scenario but also eight different decoy scenarios. The study was carried out online using a `drupal`-based survey, which communicated with **StreamingBandit** to implement the allocation of the exact positioning of the decoy. The researchers allocated participants to one of 3 between-subject conditions using **StreamingBandit**:

1. Baseline: participants in this condition were presented with a binary choice between two products, and no decoy was present. This was implemented by sending an `action` with `{"decoy":"none"}` response to the survey front-end.

2. Random: participants in this condition were presented with a random positioning of the decoy. The range of possible values of the random positioning depended on the specific scenario, and were hard-coded and retrieved using the `scenario` supplied in the context.

3. Lock-in feedback: participants in this condition were presented with a value of the decoy that depended on the previous interactions of other participants. The lock-in feedback algorithm was used to suggest a new placement each time a participant viewed a product. Subsequently, the (binary) choice made by the participant was used to update the algorithm in the `setreward` stage. We refer the reader to (Kaptein *et al.* 2016a) for details and for the exact settings of the tuning parameters.

Figure 6 presents an overview of the setup of this study. A number of the details of the implementation are covered in earlier sections of this paper: the implementation of both the baseline and the random condition are straightforward, with `self.action["decoy"] = "none"` and `self.action["decoy"] = np.random.uniform(low,high)`, respectively, as the

core `getaction` implementations. In the latter implementation the `low` and `high` bounds were implemented as a simple list indexed by the `scenario` number. Finally, the lock-in feedback condition was implemented as presented in Section 3.5, the only exception being that the `theta` was stored independently for each `scenario`. Hence, the novel part of the implementation of this study is the persistent allocation of participants to one of the three conditions; this was achieved in experiment 1 in Figure 6 by using the following `getaction` code:

```
if not("condition" in self.get_theta("user_id", self.context["user_id"])):
    self.action["note"] = "First allocation"
    draw = random.choice(["baseline", "random", "lockin"])
    self.set_theta({"condition":draw}, "userid", self.context["userid"])

self.action["condition"] =
    self.get_theta("userid", self.context["userid"])["condition"]
```

which assigns participants randomly to one of the three conditions persistently based on the `user_id` supplied in the context to the `getaction` call.[4] The data resulting from this experiment are available at https://doi.org/10.7910/DVN/FCHU0J. This field implementation provides a prime example of the use of **StreamingBandit** both for the allocation of participants to conditions in (web based-) experiments, as well as in sequential decision policies such as lock-in feedback in such experiments.

5. Conclusion and future work

This paper presented **StreamingBandit** a RESTful web application that enables researchers to develop, evaluate, and deploy CMAB policies in online experiments and field studies. By making **StreamingBandit** publicly available we hope to contribute to the more extensive use of such policies to solve statistical decision problems. The software could help in extending the currently prevailing use of basic random assignment to the use of more refined strategies throughout the social and medical sciences. To that effect, we started out with a clarification of the design rationale behind **StreamingBandit**. We explained our decision to split up the summary and the decision step of a policy – a split meant to encourage the implementation of computationally efficient online policies. We subsequently illustrated **StreamingBandit**'s versatility and flexibility in a number of examples, and we concluded with two case studies in which we used **StreamingBandit** to run field experiments.

We are currently aware of a number of limitations of **StreamingBandit**. First, as of now, **StreamingBandit** still runs single-threaded. Although parallelization for larger-scale applications ought to be relatively easy to implement on the level of policies, it may prove substantially harder within policies. Nevertheless, by forcing policies online by design, and using state-of-the-art web technology for its back end, **StreamingBandit** is already more than capable of being deployed in a multitude of small-to-medium-sized field trials. We are of the opinion that parallelization is an obvious next step in **StreamingBandit**'s development, ensuring its future scalability.

[4] Note that the actual implementation in the study differed slightly to allow for unequal sample sizes in each of the conditions. In addition, the baseline and random conditions where manually removed after sufficient data had been collected.

Second, in some applications we find that certain types of reward manifest themselves faster than others. In one instance of the use of **StreamingBandit**, for example, the decision to reject a loan to a customer after an application had been submitted to the firm was much faster than the decision (and subsequent confirmation) to accept the customer. Such an asymmetric delay might bias learning and thus needs to be addressed. Currently, we do not provide an off-the-shelf solution to this problem – admittedly because it is thus far unclear to us how to address the problem in general – hence users will need to resort to custom implementations of the `getaction` and `setreward` codes to deal with this issue.

Finally, our current CMAB libraries and toolkit still offer ample room for improvement and extension. Outside of the currently implemented methods, there are many more policies that address the exploration-exploitation trade-off in various settings. In that respect, we hope and expect the open-source nature of **StreamingBandit** to be conducive to the continued growth of the platform, encouraging researchers to implement, test, and disseminate new and existing bandit policies and algorithms.

References

Abadi M, Agarwal A, Barham P, Brevdo E, Chen Z, Citro C, Corrado GS, Davis A, Dean J, Devin M, *et al.* (2016). "**TensorFlow**: Large-Scale Machine Learning on Heterogeneous Distributed Systems." arXiv:1603.04467 [cs.LG], URL https://arxiv.org/abs/1603.04467.

Agarwal A, Bird S, Cozowicz M, Hoang L, Langford J, Lee S, Li J, Melamed D, Oshri G, Ribas O, Sen S, Slivkins A (2016). "A Multiworld Testing Decision Service." arXiv:1606.03966 [cs.LG], URL https://arxiv.org/abs/1606.03966.

Agarwal A, Hsu D, Kale S, Langford J, Li L, Schapire R (2014). "Taming the Monster: A Fast and Simple Algorithm for Contextual Bandits." In EP Xing, T Jebara (eds.), *Proceedings of the 31st International Conference on Machine Learning*, volume 32 of *Proceedings of Machine Learning Research*, pp. 1638–1646. URL http://proceedings.mlr.press/v32/agarwalb14.html.

Agrawal S, Goyal N (2012). "Analysis of Thompson Sampling for the Multi-Armed Bandit Problem." In S Mannor, N Srebro, RC Williamson (eds.), *Proceedings of the 25th Annual Conference on Learning Theory*, volume 23 of *Proceedings of Machine Learning Research*, pp. 39.1–39.26. URL http://proceedings.mlr.press/v23/agrawal12.html.

Agrawal S, Goyal N (2013). "Thompson Sampling for Contextual Bandits with Linear Payoffs." In S Dasgupta, D McAllester (eds.), *Proceedings of the 30th International Conference on Machine Learning*, volume 28 of *Proceedings of Machine Learning Research*, pp. 127–135. URL http://proceedings.mlr.press/v28/agrawal13.html.

Audibert JY, Munos R, Szepesvári C (2009). "Exploration-Exploitation Tradeoff Using Variance Estimates in Multi-Armed Bandits." *Theoretical Computer Science*, **410**(19), 1876–1902. doi:10.1016/j.tcs.2009.01.016.

Auer P, Ortner R (2010). "UCB Revisited: Improved Regret Bounds for the Stochastic Multi-Armed Bandit Problem." *Periodica Mathematica Hungarica*, **61**(1–2), 55–65. doi:10.1007/s10998-010-3055-6.

Austin PC (2011). "An Introduction to Propensity Score Methods for Reducing the Effects of Confounding in Observational Studies." *Multivariate Behavioral Research*, **46**(3), 399–424. doi:10.1080/00273171.2011.568786.

Bastani H, Bayati M (2020). "Online Decision-Making with High-Dimensional Covariates." *Operations Research*, **68**(1), 276–294. doi:10.1287/opre.2019.1902.

Berry DA, Fristedt B (1985). *Bandit Problems: Sequential Allocation of Experiments*. Springer-Verlag. doi:10.1007/978-94-015-3711-7.

Beygelzimer A, Hsu DJ, Langford J, Zhang T (2010). "Agnostic Active Learning without Constraints." In JD Lafferty, CKI Williams, J Shawe-Taylor, RS Zemel, A Culotta (eds.), *Advances in Neural Information Processing Systems 23*, pp. 199–207. Curran Associates. URL http://papers.nips.cc/paper/4014-agnostic-active-learning-without-constraints.pdf.

Bubeck S, Munos R, Stoltz G (2011). "Pure Exploration in Finitely-Armed and Continuous-Armed Bandits." *Theoretical Computer Science*, **412**(19), 1832–1852. doi:10.1016/j.tcs.2010.12.059.

Chapelle O, Li L (2011). "An Empirical Evaluation of Thompson Sampling." In J Shawe-Taylor, RS Zemel, PL Bartlett, F Pereira, KQ Weinberger (eds.), *Advances in Neural Information Processing Systems 24*, pp. 2249–2257. Curran Associates. URL http://papers.nips.cc/paper/4321-an-empirical-evaluation-of-thompson-sampling.pdf.

Collet J, Sassolas T, Lhuillier Y, Sirdey R, Carlier J (2016). "Leveraging Distributed **GraphLab** for Program Trace Analysis." In *2016 International Conference on High Performance Computing & Simulation (HPCS)*. doi:10.1109/hpcsim.2016.7568341.

Cui H, Zhang H, Ganger GR, Gibbons PB, Xing EP (2016). "**GeePS**: Scalable Deep Learning on Distributed GPUs with a GPU-Specialized Parameter Server." In *Proceedings of the Eleventh European Conference on Computer Systems*. doi:10.1145/2901318.2901323.

De B (2017). "API Testing Strategy." In *API Management: An Architect's Guide to Developing and Managing APIs for Your Organization*, pp. 153–164. Apress. doi:10.1007/978-1-4842-1305-6_9.

Dudík M, Erhan D, Langford J, Li L (2012). "Sample-Efficient Nonstationary Policy Evaluation for Contextual Bandits." In *Proceedings of the Twenty-Eighth Conference on Uncertainty in Artificial Intelligence*, pp. 247–254. URL http://dl.acm.org/citation.cfm?id=3020652.3020681.

Dudík M, Hsu DJ, Kale S, Karampatziakis N, Langford J, Reyzin L, Zhang T (2011a). "Efficient Optimal Learning for Contextual Bandits." arXiv:1106.2369 [cs.LG], URL https://arxiv.org/abs/1106.2369.

Dudík M, Langford J, Li L (2011b). "Doubly Robust Policy Evaluation and Learning." In *Proceedings of the 28th International Conference on International Conference on Machine Learning*, pp. 1097–1104. URL http://dl.acm.org/citation.cfm?id=3104482.3104620.

Eckles D, Kaptein M (2014). "Thompson Sampling with the Online Bootstrap." arXiv:1410.4009 [cs.LG], URL https://arxiv.org/abs/1410.4009.

Galbraith B (2016). *Python Library for Multi-Armed Bandits*. Github Repository, URL https://github.com/bgalbraith/bandits.

Garivier A, Cappé O (2011). "The KL-UCB Algorithm for Bounded Stochastic Bandits and Beyond." In SM Kakade, U von Luxburg (eds.), *Proceedings of the 24th Annual Conference on Learning Theory*, volume 19 of *Proceedings of Machine Learning Research*, pp. 359–376. URL http://proceedings.mlr.press/v19/garivier11a.html.

Google (2018). ***Optimize***, *Google Analytics Solutions*. URL https://www.google.com/analytics/optimize/features/.

Hanneke S (2014). "Theory of Disagreement-Based Active Learning." *Foundations and Trends® in Machine Learning*, **7**(2–3), 131–309. doi:10.1561/2200000037.

Hido S, Tokui S, Oda S (2013). "**Jubatus**: An Open Source Platform for Distributed Online Machine Learning." In *NIPS 2013 Workshop on Big Learning, Lake Tahoe*.

Ierusalimschy R (2016). *Programming in Lua*. 4th edition. Lua.org.

Imbens GW, Rubin DB (2015). *Causal Inference for Statistics, Social, and Biomedical Sciences*. Cambridge University Press. doi:10.1017/cbo9781139025751.

Jiang B, Shi X, Shang H, Geng Z, Glass A (2016). "A Framework for Network AB Testing." arXiv:1610.07670 [stat.AP], URL https://arxiv.org/abs/1610.07670.

Kaptein M (2014). "The Use of Thompson Sampling to Increase Estimation Precision." *Behavior Research Methods*, **47**(2), 409–423. doi:10.3758/s13428-014-0480-0.

Kaptein M, Ianuzzi D (2016). "Lock in Feedback in Sequential Experiments." arXiv:1502.00598 [cs.LG], URL https://arxiv.org/abs/1502.00598.

Kaptein MC, Emden RV, Iannuzzi D (2016a). "Tracking the Decoy: Maximizing the Decoy Effect Through Sequential Experimentation." *Palgrave Communications*, **2**(1). doi:10.1057/palcomms.2016.82.

Kaptein MC, Van Emden R, Iannuzzi D (2016b). "Investigation of the Concept of Beauty via a Lock-in Feedback Experiment." arXiv:1607.08108 [physics.soc-ph], URL https://arxiv.org/abs/1607.08108.

Kaufmann E, Cappé O, Garivier A (2012a). "**pymaBandits**: MATLAB and Python Packages for Multi-Armed Bandits." URL http://mloss.org/software/view/415/.

Kaufmann E, Korda N, Munos R (2012b). "Thompson Sampling: An Asymptotically Optimal Finite-Time Analysis." In NH Bshouty, G Stoltz, N Vayatis, T Zeugmann (eds.), *Algorithmic Learning Theory*, Lecture Notes in Computer Science, pp. 199–213. Springer-Verlag. doi:10.1007/978-3-642-34106-9_18.

Klenov K (2015). *Python's Web Framework Benchmarks*. Github Repository, URL http://klen.github.io/py-frameworks-bench/.

Komiyama J, Honda J, Nakagawa H (2015). "Optimal Regret Analysis of Thompson Sampling in Stochastic Multi-Armed Bandit Problem with Multiple Plays." In F Bach, D Blei (eds.), *Proceedings of the 32nd International Conference on Machine Learning*, volume 37 of *Proceedings of Machine Learning Research*, pp. 1152–1161. URL http://proceedings.mlr.press/v37/komiyama15.html.

Lachin JM, Matts JP, Wei LJ (1988). "Randomization in Clinical Trials: Conclusions and Recommendations." *Controlled Clinical Trials*, **9**(4), 365–374. doi:10.1016/0197-2456(88)90049-9.

Langford J, Li L, Strehl A (2011). ***Vowpal Wabbit***. Github Repository, URL https://github.com/JohnLangford/vowpal_wabbit/wiki.

Li L, Chu W, Langford J, Schapire RE (2010). "A Contextual-Bandit Approach to Personalized News Article Recommendation." In *Proceedings of the 19th International Conference on World Wide Web*. doi:10.1145/1772690.1772758.

Li L, Chu W, Langford J, Wang X (2011). "Unbiased Offline Evaluation of Contextual-Bandit-Based News Article Recommendation Algorithms." In *Proceedings of the Fourth ACM International Conference on Web Search and Data Mining*. doi:10.1145/1935826.1935878.

Macready WG, Wolpert DH (1998). "Bandit Problems and the Exploration/Exploitation Tradeoff." *IEEE Transactions on Evolutionary Computation*, **2**(1), 2–22. doi:10.1109/4235.728210.

Mandelbaum A (1987). "Continuous Multi-Armed Bandits and Multiparameter Processes." *The Annals of Probability*, **15**(4), 1527–1556. doi:10.1214/aop/1176991992.

Meng X, Bradley J, Yavuz B, Sparks E, Venkataraman S, Liu D, Freeman J, Tsai D, Amde M, Owen S, Xin D, Xin R, Franklin MJ, Zadeh R, Zaharia M, Talwalkar A (2016). "**MLlib**: Machine Learning in Apache Spark." *Journal of Machine Learning Research*, **17**(34), 1–7.

Merkel D (2014). "**Docker**: Lightweight Linux Containers for Consistent Development and Deployment." *Linux Journal*, **2014**(239), 2.

Michalak S, DuBois A, DuBois D, Wiel SV, Hogden J (2012). "Developing Systems for Real-Time Streaming Analysis." *Journal of Computational and Graphical Statistics*, **21**(3), 561–580. doi:10.1080/10618600.2012.657144.

Mixpanel (2017). *Mixpanel: Product Analytics for Product People*. URL https://mixpanel.com.

Nugent T (2015). *Epsilon-Greedy, Softmax and LinUCB Contextual Bandit Implementations*. Github Repository, URL https://github.com/timnugent/bandit-algorithms.

Optimizely (2017). *Optimizely*. URL https://www.optimizely.com/.

Osban I, Roy BV (2015). "Bootstrapped Thompson Sampling and Deep Exploration." arXiv:1507.00300 [stat.ML], URL https://arxiv.org/abs/1507.00300.

Owen AB, Eckles D (2012). "Bootstrapping Data Arrays of Arbitrary Order." *The Annals of Applied Statistics*, **6**(3), 895–927. doi:10.1214/12-aoas547.

Pandey S, Chakrabarti D, Agarwal D (2007). "Multi-Armed Bandit Problems with Dependent Arms." In *Proceedings of the 24th International Conference on Machine Learning*. doi:10.1145/1273496.1273587.

Pearl J (2009). *Causality*. Cambridge University Press. doi:10.1017/cbo9780511803161.

Reagen B, Whatmough P, Adolf R, Rama S, Lee H, Lee SK, Hernandez-Lobato JM, Wei GY, Brooks D (2016). "**Minerva**: Enabling Low-Power, Highly-Accurate Deep Neural Network Accelerators." In *2016 ACM/IEEE 43rd Annual International Symposium on Computer Architecture (ISCA)*. doi:10.1109/isca.2016.32.

Schwartz EM, Bradlow ET, Fader PS (2017). "Customer Acquisition via Display Advertising Using Multi-Armed Bandit Experiments." *Marketing Science*, **36**(4), 500–522. doi:10.1287/mksc.2016.1023.

Seide F, Agarwal A (2016). "**CNTK**: Microsoft's Open-Source Deep-Learning Toolkit." In *Proceedings of the 22nd ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*. doi:10.1145/2939672.2945397.

Sola D (2015). *Bandit Algorithms and Test Framework in Java*. Github Repository, URL https://github.com/danisola/bandit.

Striatum Contributors (2016). *Contextual Bandit in Python*. Github Repository, URL https://github.com/ntucllab/striatum.

Sutton RS, Barto AG (1998). *Reinforcement Learning: An Introduction*. MIT Press.

Szepesvári C (2010). "Algorithms for Reinforcement Learning." *Synthesis Lectures on Artificial Intelligence and Machine Learning*, **4**(1), 1–103. doi:10.2200/s00268ed1v01y201005aim009.

Tene G (2015). ***wrk2**: A Constant Throughput, Correct Latency Recording Variant of **wrk***. Github Repository, URL https://github.com/giltene/wrk2.

The **iPerf** Authors (2016). ***iPerf** – The Ultimate Speed Test Tool for TCP, UDP and SCTP*. URL https://iperf.fr/.

The **Tornado** Authors (2016). ***Tornado** Software Version 4.4.2*. URL http://www.tornadoweb.org.

Thompson WR (1933). "On the Likelihood That One Unknown Probability Exceeds Another in View of the Evidence of Two Samples." *Biometrika*, **25**(3/4), 285–294. doi:10.2307/2332286.

Van Rossum G, *et al.* (2011). *Python Programming Language*. URL https://www.python.org/.

Villar SS, Bowden J, Wason J (2015). "Multi-Armed Bandit Models for the Optimal Design of Clinical Trials: Benefits and Challenges." *Statistical Science*, **30**(2), 199–215. doi:10.1214/14-sts504.

Welford BP (1962). "Note on a Method for Calculating Corrected Sums of Squares and Products." *Technometrics*, **4**(3), 419–420. doi:10.2307/1266577.

Whittle P (1980). "Multi-Armed Bandits and the Gittins Index." *Journal of the Royal Statistical Society B*, **42**(2), 143–149. doi:10.1111/j.2517-6161.1980.tb01111.x.

Yelp (2014). ***MOE**: Metric Optimization Engine*. Github Repository, URL https://github.com/Yelp/MOE.

A. Setting up an experiment

This appendix introduces the front-end of **StreamingBandit**.[5] We will how show to get from the login screen to setting up your first simulation using one of the default experiments.

First, when you have set up the front-end (using, e.g., the available **Docker** container), go to the login screen in your browser (for the **Docker** container this would be `http://localhost` or the **Docker**-set IP address) as shown in Figure 7.

After logging in, you will find the dashboard as in Figure 8. To show all the active experiments, click on `Experiments`. This will bring you to an environment as shown in Figure 10. Continue clicking on the `Create` button, which will give you an empty `Create Experiment` field, as in Figure 10.

On the creation page you can fill in a name, for example `E-First`, and select a default experiment from the `Use experiment template` list. Selecting the default ϵ-first experiment, will end up with a filled-in form, as in Figure 11. Next, clicking on the `Save` button will save the experiment in the database.

When the experiment has been created the dashboard (Figure 12) shows that the experiment is active and has an ID and key assigned. Clicking on the `Edit` button will take you back to the settings of the experiment. Now you can choose to go to the `Simulate` tab as displayed in Figure 13. After filling in 1000 for the number of iterations and 43123 as the seed you can click `Run a simulation of the experiment`, which will give a result as in Figure 14. Finally, you can click on the `Theta` tab and inspect the parameters that are stored in the database (Figure 15). Here you can also download the data that has been logged for the current experiment.

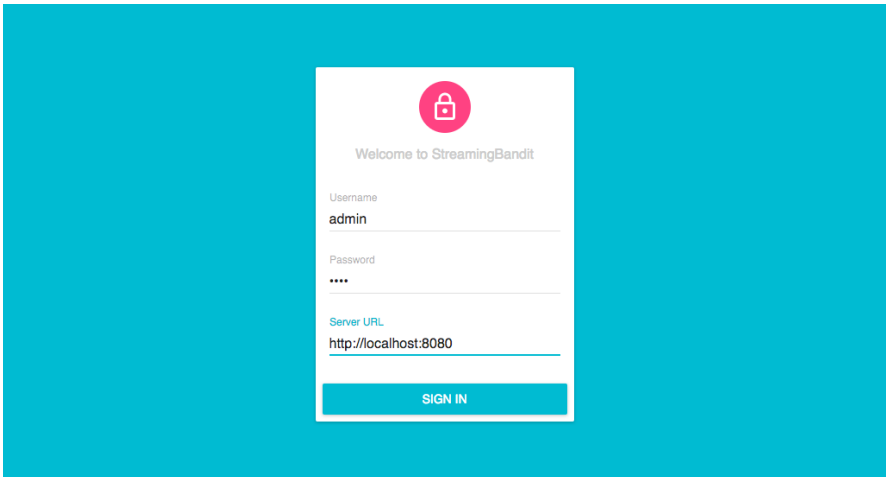

Figure 7: The login screen. Here you can set the URL for the back-end and login.

[5] Which can be found at `https://github.com/Nth-iteration-labs/streamingbandit-ui`.

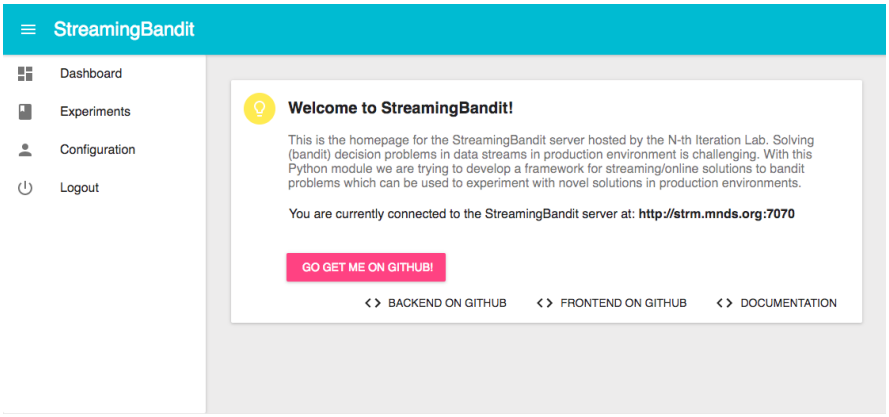

Figure 8: The dashboard of **StreamingBandit**, here you can navigate to the list of experiments and find some extra information.

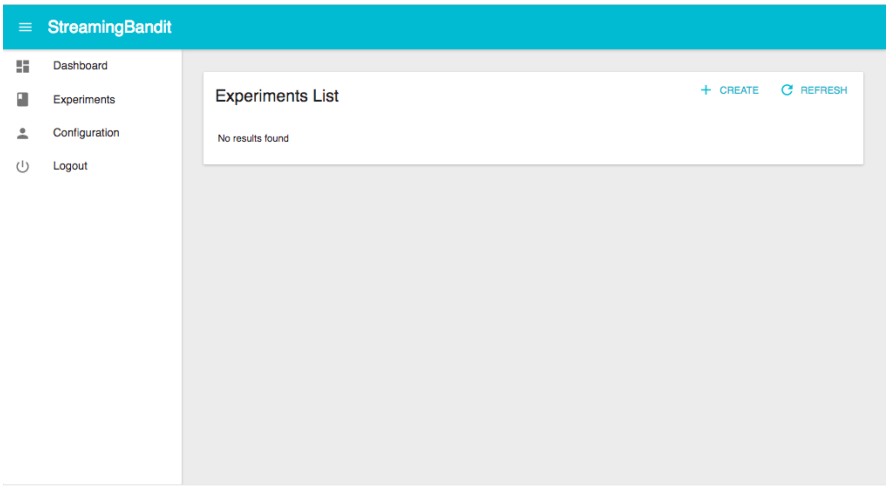

Figure 9: This screenshot shows an empty list of experiments. You can start creating an experiment by clicking on the `Create` button.

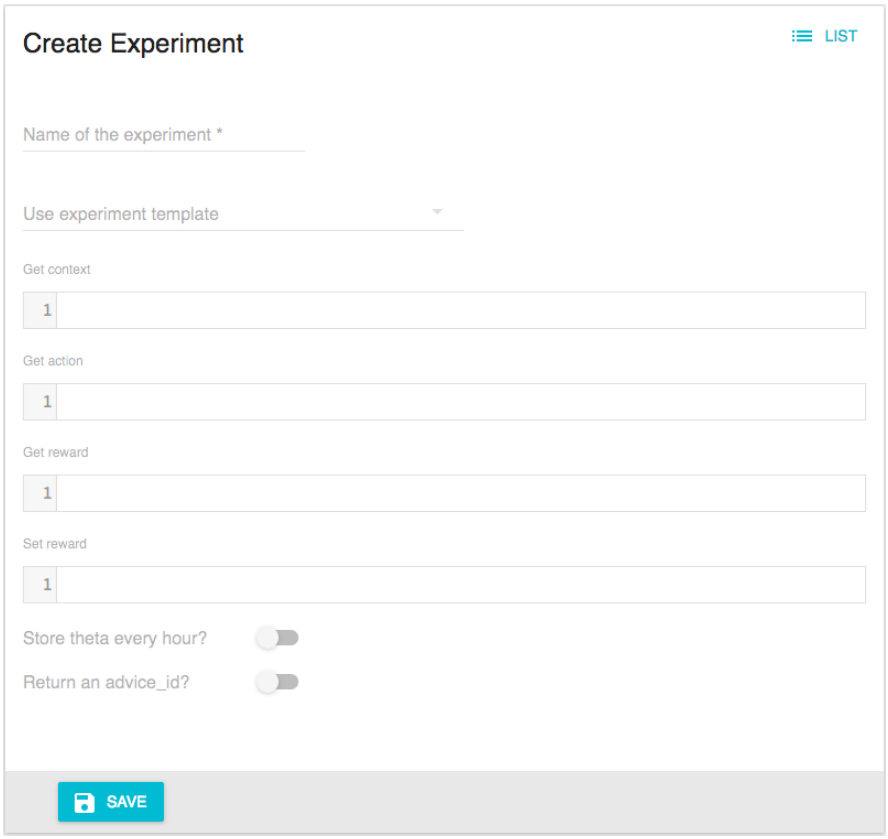

Figure 10: This is an empty form for an experiment, which is normally shown inside the dashboard. Here you can type a name, choose a default and edit the code for the experiment.

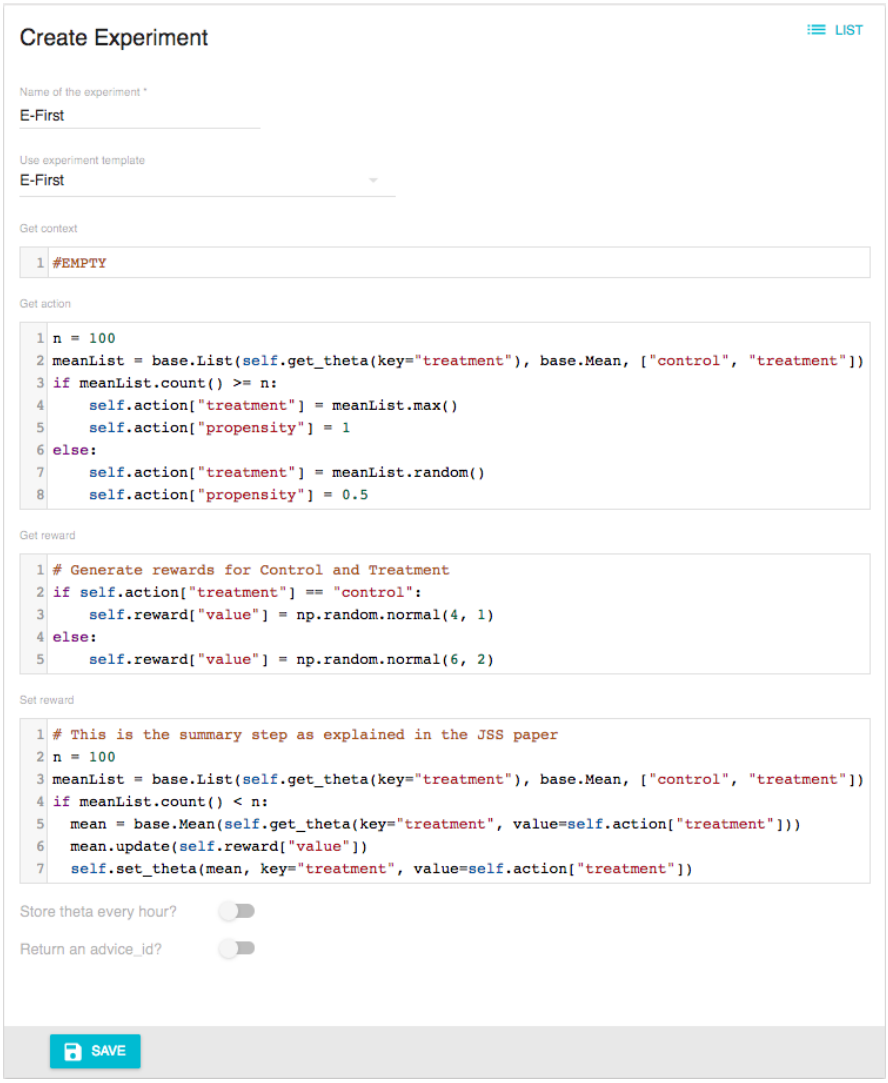

Figure 11: We have selected the E-First template, which is automatically filled in the correct fields. Pressing the `Save` button will create the experiment.

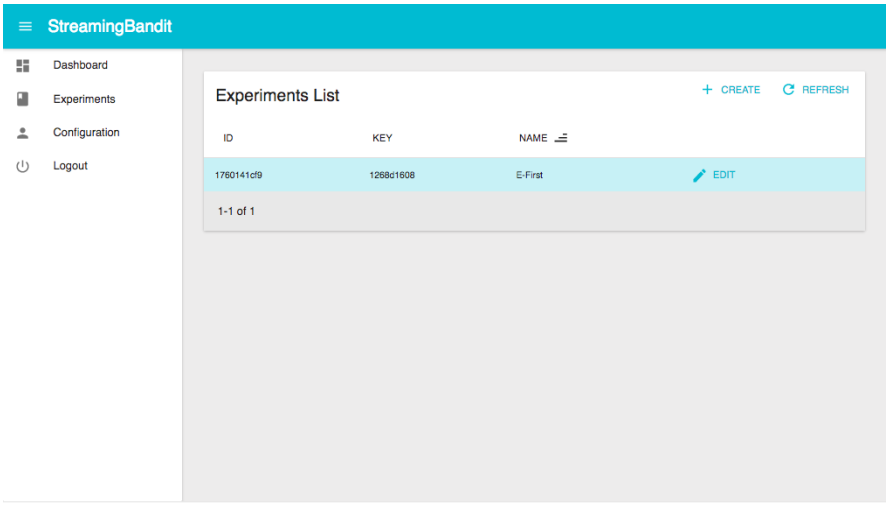

Figure 12: If you return to the dashboard you can view and edit your newly created experiment (and the associated ID and key).

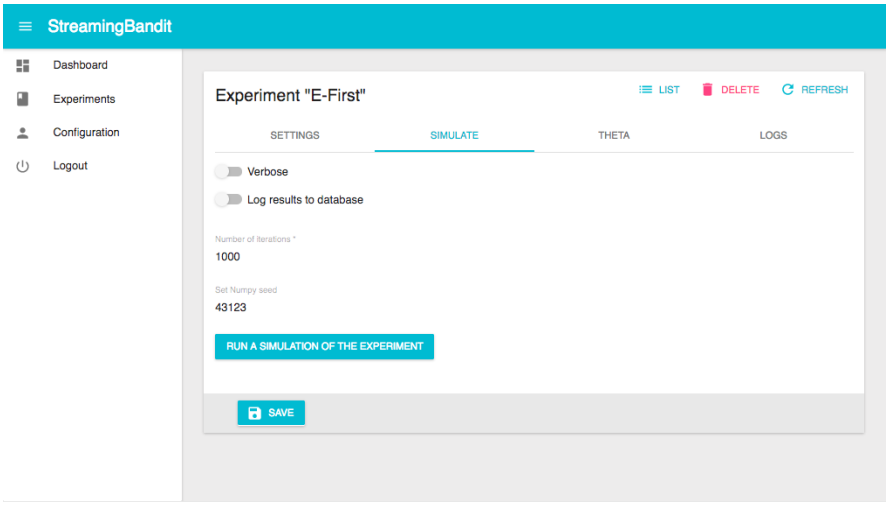

Figure 13: Here you can see the simulation panel, which you can use to easily run a simulation of the experiment. You can set the speed and the the number of iterations, log the results to the database and even show verbose results. Click `Run a simulation of the experiment` to run a simulation and get an output of the results.

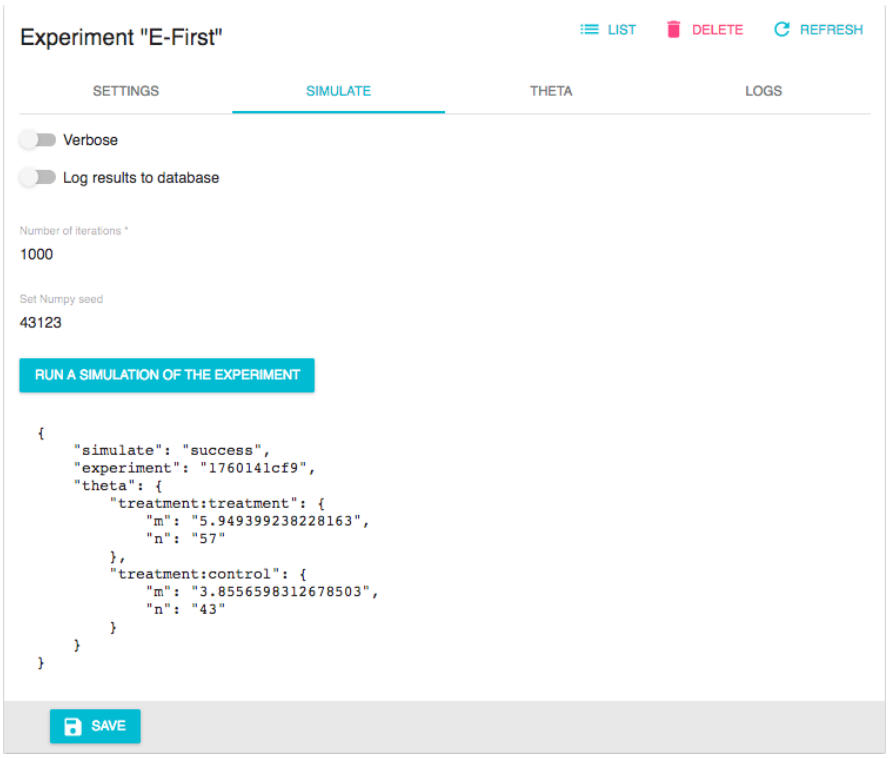

Figure 14: Run a simulation and look at the output of the results.

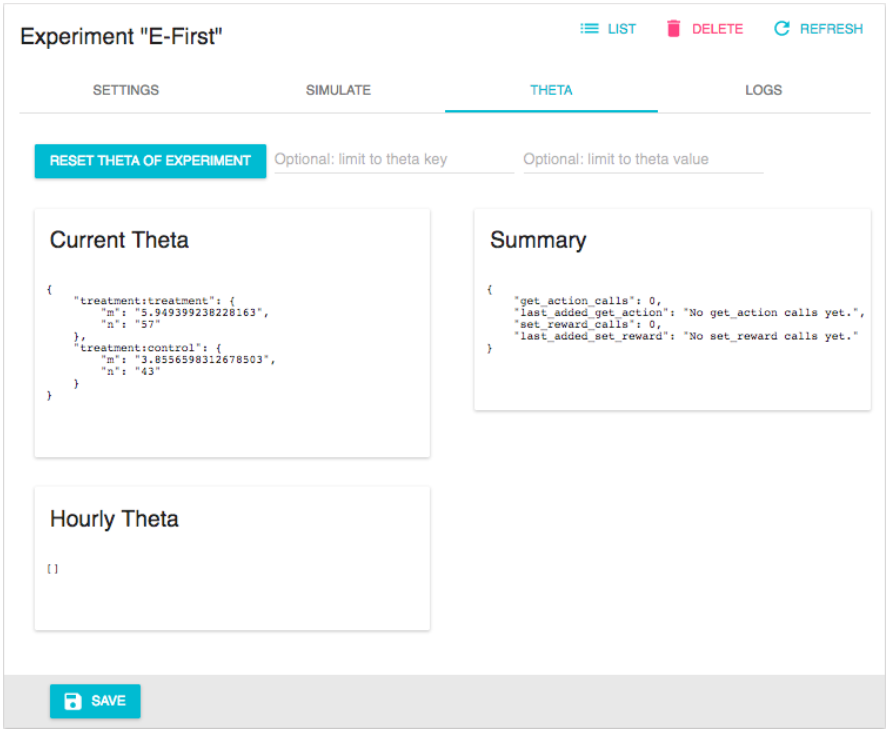

Figure 15: The theta panel shows the current state of θ and other information.

Affiliation:

Jules M.A. Kruijswijk
Tilburg University
Statistics and Research Methods
Prof. Cobbenhagenlaan 225
Simon Building, room S 719
5037 DB Tilburg, Netherlands
E-mail: j.m.a.kruijswijk@tilburguniversity.edu

Robin van Emden, Maurits C. Kaptein
Jheronimus Academy of Data Science
Sint Janssingel 92
5211 DA 's-Hertogenbosch, Netherlands
E-mail: r.a.vanemden@tilburguniversity.edu, m.c.kaptein@tilburguniversity.edu

Petri Parvinen
University of Helsinki
Business and Management
Latokartanonkaari 7
00014 University of Helsinki, Finland
E-mail: petri.parvinen@helsinki.fi

Journal of Statistical Software http://www.jstatsoft.org/
published by the Foundation for Open Access Statistics http://www.foastat.org/
August 2020, Volume 94, Issue 9 *Submitted:* 2016-10-21
doi:10.18637/jss.v094.i09 *Accepted:* 2018-09-06